Reading 5: Optimal Allocation of Flows and the Transportation Problem

Antonio Páez
My Name

14 October, 2021

Introduction

NOTE: This is an R Markdown Notebook. When you execute code within the notebook, the results appear beneath the code.

We have seen in previous readings how `R` can effectively handle many data analysis operations by means of data tables and data operations. We have also used a variety of geographic information tools for processing and analyzing data. Data analysis can be made more powerful by introducing elements of control flow.

Control flow is the order in which instructions are executed in a program. An important category of control flow is a *conditional branch*, two forms of which we will introduce here:

- Choice: Execute a set of statements *only if* a certain condition is met.
- Loop: Execute a set of statements zero or more times *until* some condition is met.

In this tutorial, we will introduce conditional branching using the R statistical computing language.

To use this note you will need the following:

- The following data objects (available in your course package)
  - `hamilton_graph`

You have already used the graph object (`hamilton_graph`) in Readings 3 and 4, and you will recall that it includes the road network in Hamilton sourced from Open Street Maps. The graph has all roads, including provincial highways (e.g., the 403), and some nodes are labeled according to the zoning system used for the analysis of transportation data in Hamilton.

Next, we load the packages needed for analysis (some may need to be installed before they are available for loading):

```
library(envsocty3LT3) # Course package
library(igraph) # Package for network analysis
library(scales) # This package provides scales functions for visualization
library(sf) # Package to work with geospatial information in simple features format
library(tidygraph) # Package to work with spatial networks
library(tidyverse) # Family of packages for data manipulation, analysis, and visualization
library(units) # Package for working with units in `R`
```

Background

One way of motivating the transportation problem is as follows.

We have seen how travelers respond to differences in the cost of transportation (think the distance-decay effect in spatial interaction). This effect implies that travelers display a preference for destinations that are less costly to reach.

Imagine now the perspective of businesses who must compete for consumers. The classical example is the ice cream vendors on a long stretch of beach (this example is due to American statistician Harold Hotelling). This simple example has the following conditions:

1. The cost of relocation is zero. An ice-cream vendor can just move their cart to a new location if they wish to do so.
2. We can treat the world as a line (beach-goers are all along the beach), which means that the problem has only one spatial dimension.
3. Beach-goers, that is, potential consumers of ice-cream, are located at random along the beach, which is to say that their density is uniform (every segment of beach has more or less the same number of customers.)
4. Beach-goers prefer to spend less time walking towards an ice-cream cart than more, and more time sunbathing than walking to the ice-cream cart. In other words, they are sensitive to the effort needed to get an ice-cream.

Under these conditions, we can set up the following situation.

The initial location of the two ice-cream vendors is at points 2.5 and 7.5 along the beach. Both vendors offer ice-cream of identical quality, at $5 dollars a serving. Finally, the beach is homogeneous, so the effort needed to reach the ice cream vendors is the same irrespective of direction. These conditions can be expressed as follows:

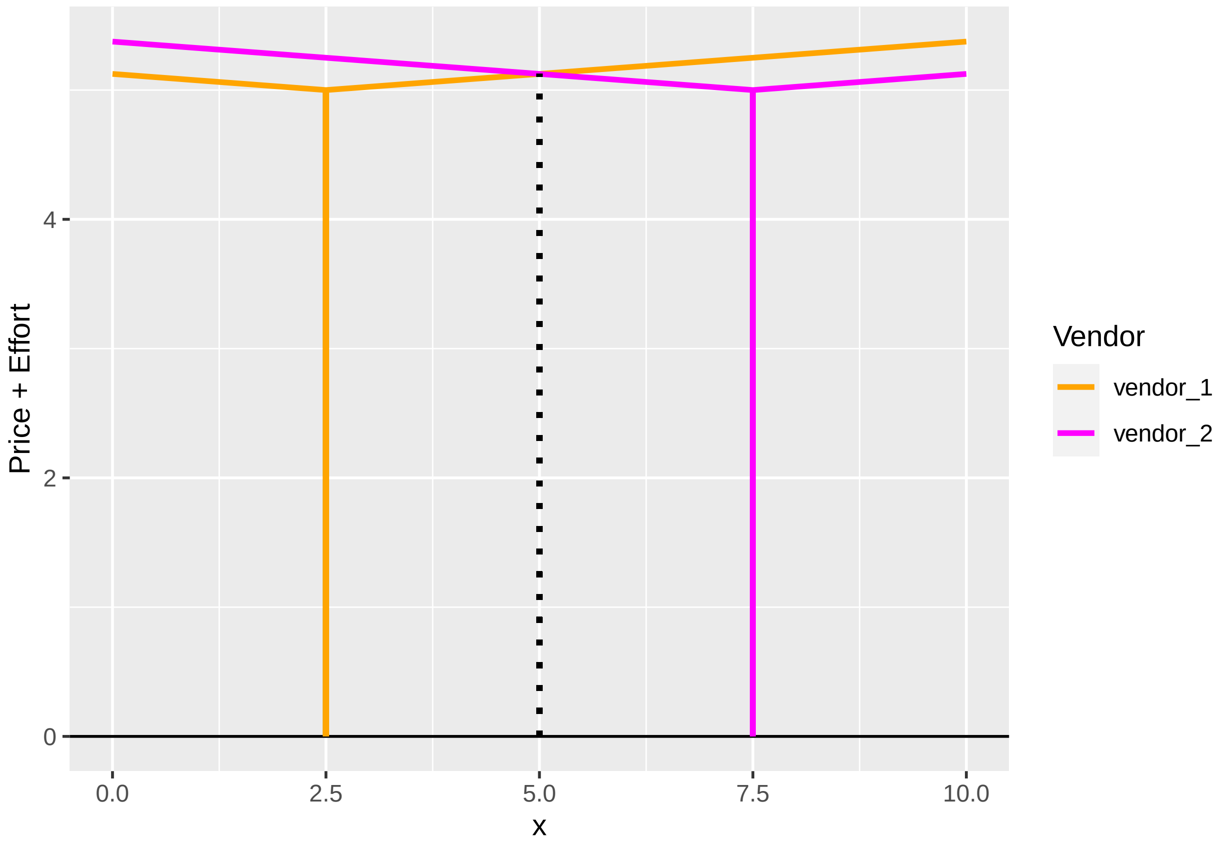
```
location_1 = 2.5 # Location of vendor 1 (this must always be the one on the left)
location_2 = 7.5 # Location of vendor 2 (this must always be the one on the right)
price_1 = 5 # Price of ice-cream of vendor 1
price_2 = 5 # Price of ice-cream of vendor 2
delivery_1 = 0.05 # This is the effort needed to reach ice-cream vendor 1 (on the left)
delivery_2 = 0.05 # This is the effort needed to reach ice-cream vendor 2 (on the right)
```

We can create a data frame with the conditions above, call it `market_areas`. We will `mutate()` the data frame to add two new columns, `market_1` and `market_2`, which are the price of the ice-cream plus the effort of reaching an ice-cream cart:

```
market_areas <- data.frame(x = seq(from = 0, to = 10, by = 0.1),
                           price_1 = price_1,
                           price_2 = price_2,
                           delivery_1 = delivery_1,
                           delivery_2 = delivery_2,
                           location_1 = location_1,
                           location_2 = location_2) %>%
  mutate(vendor_1 = price_1 + abs(location_1 - x) * delivery_1, # Market area 1 depends on the location
         vendor_2 = price_2 + abs(location_2 - x) * delivery_2) # Market area 2 depends on the location
```

We can plot the market areas in this way. The horizontal line is the beach (where customers are distributed randomly). The vertical axis is a combination of the price of the ice-cream plus the effort of reaching the carts:

```
market_areas %>% # Pass market areas to the next function
  select(x, # Select three variables from the data frame: `x`, `vendor_1`, and `vendor_2`
         vendor_1,
         vendor_2) %>%
  pivot_longer(cols = starts_with("vendor"), # Pivot longer so that each row is point in space and the
               names_to = "Vendor",
               values_to = "Price + Effort") %>%
  mutate(Market = factor(Vendor, # Convert the variable `Vendor` to a factor with labels `Vendor 1` and
                         levels = c("vendor_1", "vendor_2"),
                         labels = c("Vendor 1", "Vendor 2"))) %>%
  ggplot(aes(x = x, y = `Price + Effort`, color = Vendor)) + # Create a ggplot object; the x-axis is th
  geom_line(size = 1) + # Plot the total price + effort as lines
  geom_hline(yintercept = 0,  # Plot a horizontal line: this is the beach
             color = "black") +
  geom_segment(x = location_1, # Plot a segment to indicate the price of the ice-cream offered by vendo
               xend = location_1,
               y = 0,
               yend = price_1,
               color = "orange", # The color of this line is orange
               size = 1) +
  geom_segment(x = location_2, # Plot a segment to indicate the price of the ice-cream offered by vendo
               xend = location_2,
               y = 0,
               yend = price_2,
               color = "magenta", # The color of this line is magenta
               size = 1) +
  geom_segment(x = (price_2 - price_1 + location_1 * delivery_1 + location_2 * delivery_2)/(delivery_1 +
               xend = (price_2 - price_1 + location_1 * delivery_1 + location_2 * delivery_2)/(delivery_
               y = 0,
               yend = price_2 + (location_2 - (price_2 - price_1 + location_1 * delivery_1 + location_2
               color = "black",
               linetype = "dotted",
               size = 1) +
  scale_color_manual(values = c("vendor_1" = "orange", "vendor_2" = "magenta")) # Set the colors of the
```



We see that there is an indifference point where the lines of price-plus-effort cross. To the left of that point, people start preferring the orange cart, and to the right the magenta cart. The point of indifference is exactly in the middle of the beach, which means that the vendors split the market equally, 50-50. The point of indifference can be found for this simple example by finding the location where the price (plus effort) is identical for both ice-cream vendors. We will call the price plus effort p (you can think of the effort as the value of the time needed to reach the cart, or as the cost of delivering the ice-cream). So, the total is p_1 for vendor 1 and p_2 for vendor 2. The point of indifference is the location where:

$$p_1 = p_2$$

Since p depends on the *base price* (the 5 dollars of the ice-cream) and the effort depends on the location of vendor 1 (x_1) and the location of vendor 2 x_2, we have that the total prices for the vendors are:

$$p_1 = b_1 + (x - x_1)d_1 p_2 = b_2 + (x_2 - x)d_2$$

where x is the point where the two prices are identical. We can solve for x as follows:

$$b_1 + (x - x_1)d_1 = b_2 + (x_2 - x)d_2(x - x_1)d_1 - (x_2 - x) * d_2 = b_2 - b_1 x d_1 - x_1 d_1 - x_2 d_2 + x d_2 = b_2 - b_1 x d_1 + x d_2 = b_2 - b_1 + x_1 d_1 + x_2 d_2 x(d$$

This means that the point of indifference is:

$$x = \frac{b_2 - b_1 + x_1 d_1 + x_2 d_2}{d_1 + d_2}$$

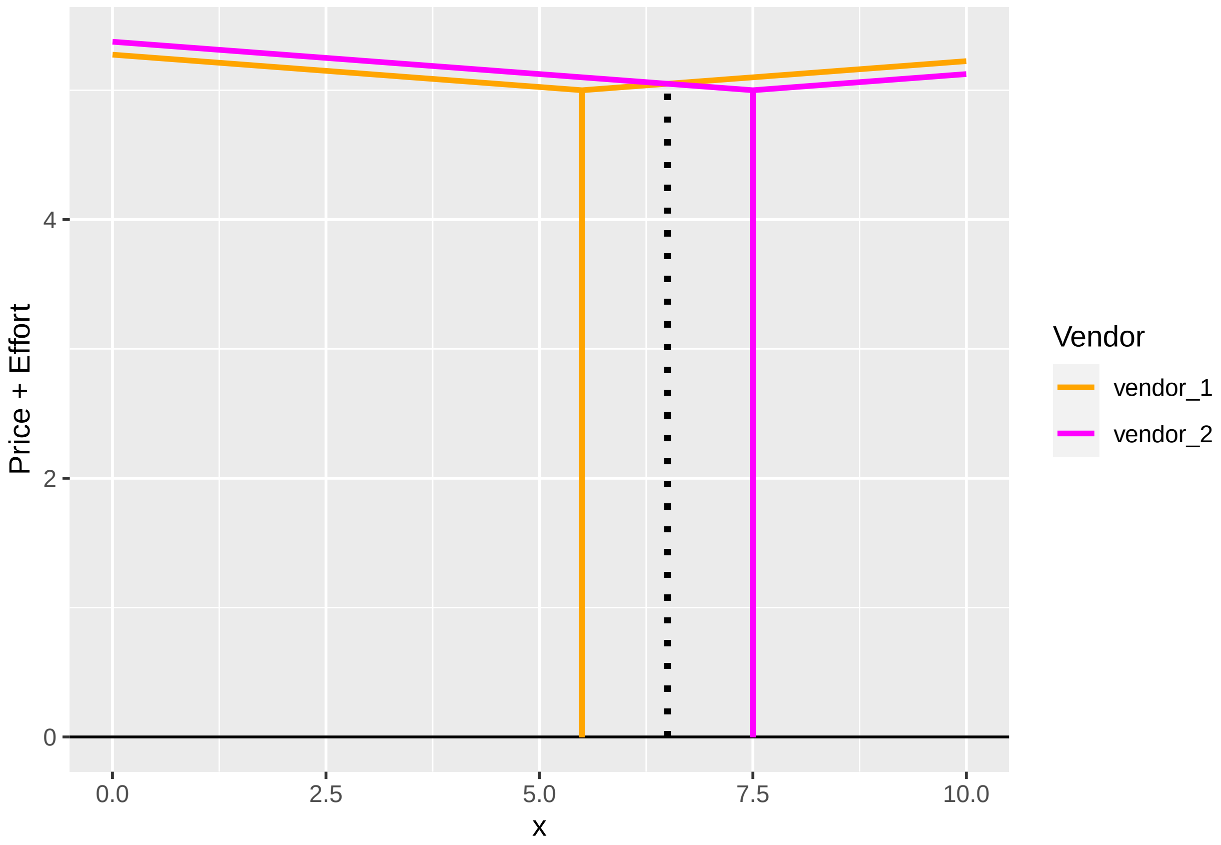
Let us say that one of the two vendors (the orange one) becomes a little greedy, and tries to capture a bigger share of the market. They can do this by relocating. Change the conditions here:

```r
location_1 = 5.5 # Location of vendor 1 (this must always be the one on the left)
location_2 = 7.5 # Location of vendor 2 (this must always be the one on the right)
price_1 = 5 # Price of ice-cream of vendor 1
price_2 = 5 # Price of ice-cream of vendor 2
delivery_1 = 0.05 # This is the effort needed to reach ice-cream vendor 1 (on the left)
delivery_2 = 0.05 # This is the effort needed to reach ice-cream vendor 2 (on the right)
```

Vendor 1 has moved to location 5.5. Recreate the data frame with these new conditions, and plot the market areas:

```r
market_areas <- data.frame(x = seq(from = 0, to = 10, by = 0.1),
                           price_1 = price_1,
                           price_2 = price_2,
                           delivery_1 = delivery_1,
                           delivery_2 = delivery_2,
                           location_1 = location_1,
                           location_2 = location_2) %>%
  mutate(vendor_1 = price_1 + abs(location_1 - x) * delivery_1, # Market area 1 depends on the location
         vendor_2 = price_2 + abs(location_2 - x) * delivery_2) # Market area 2 depends on the location

market_areas %>% # Pass market areas to the next function
  select(x, # Select three variables from the data frame: `x`, `vendor_1`, and `vendor_2`
         vendor_1,
         vendor_2) %>%
  pivot_longer(cols = starts_with("vendor"), # Pivot longer so that each row is point in space and the
               names_to = "Vendor",
               values_to = "Price + Effort") %>%
  mutate(Market = factor(Vendor, # Convert the variable `Vendor` to a factor with labels `Vendor 1` and
                         levels = c("vendor_1", "vendor_2"),
                         labels = c("Vendor 1", "Vendor 2"))) %>%
  ggplot(aes(x = x, y = `Price + Effort`, color = Vendor)) + # Create a ggplot object; the x-axis is th
  geom_line(size = 1) + # Plot the total price + effort as lines
  geom_hline(yintercept = 0,  # Plot a horizontal line: this is the beach
             color = "black") +
  geom_segment(x = location_1, # Plot a segment to indicate the price of the ice-cream offered by vendo
               xend = location_1,
               y = 0,
               yend = price_1,
               color = "orange", # The color of this line is orange
               size = 1) +
  geom_segment(x = location_2, # Plot a segment to indicate the price of the ice-cream offered by vendo
               xend = location_2,
               y = 0,
               yend = price_2,
               color = "magenta", # The color of this line is magenta
               size = 1) +
  geom_segment(x = (price_2 - price_1 + location_1 * delivery_1 + location_2 * delivery_2)/(delivery_1 +
               xend = (price_2 - price_1 + location_1 * delivery_1 + location_2 * delivery_2)/(delivery_
               y = 0,
               yend = price_2 + (location_2 - (price_2 - price_1 + location_1 * delivery_1 + location_2
               color = "black",
               linetype = "dotted",
               size = 1) +
  scale_color_manual(values = c("vendor_1" = "orange", "vendor_2" = "magenta")) # Set the colors of the
```



By relocating, vendor 1 has pushed the point of indifference and captured a bigger market area, since now more people along the beach prefer their cart to the magenta ice-cream cart. This strategy of relocation gives raise to a dynamic process of tit-for-tat, since the magenta cart can relocate as well. Hotelling's insight is that the equilibrium situation with two vendors is when both are located in the middle of the beach and sharing the market 50-50. Neither has an incentive to move: the first one to move ends up with a smaller market share (you can experiment with this using the plots above; just remember, whichever cart is on the left automatically becomes `Vendor 1`). Unfortunately for beach-goers, the situation with two vendors at the center of the beach is not as convenient as two vendors at 2.5 and 7.5: some of them, those who are closest to the end of the beach, now have to walk a longer distance to get ice-cream!

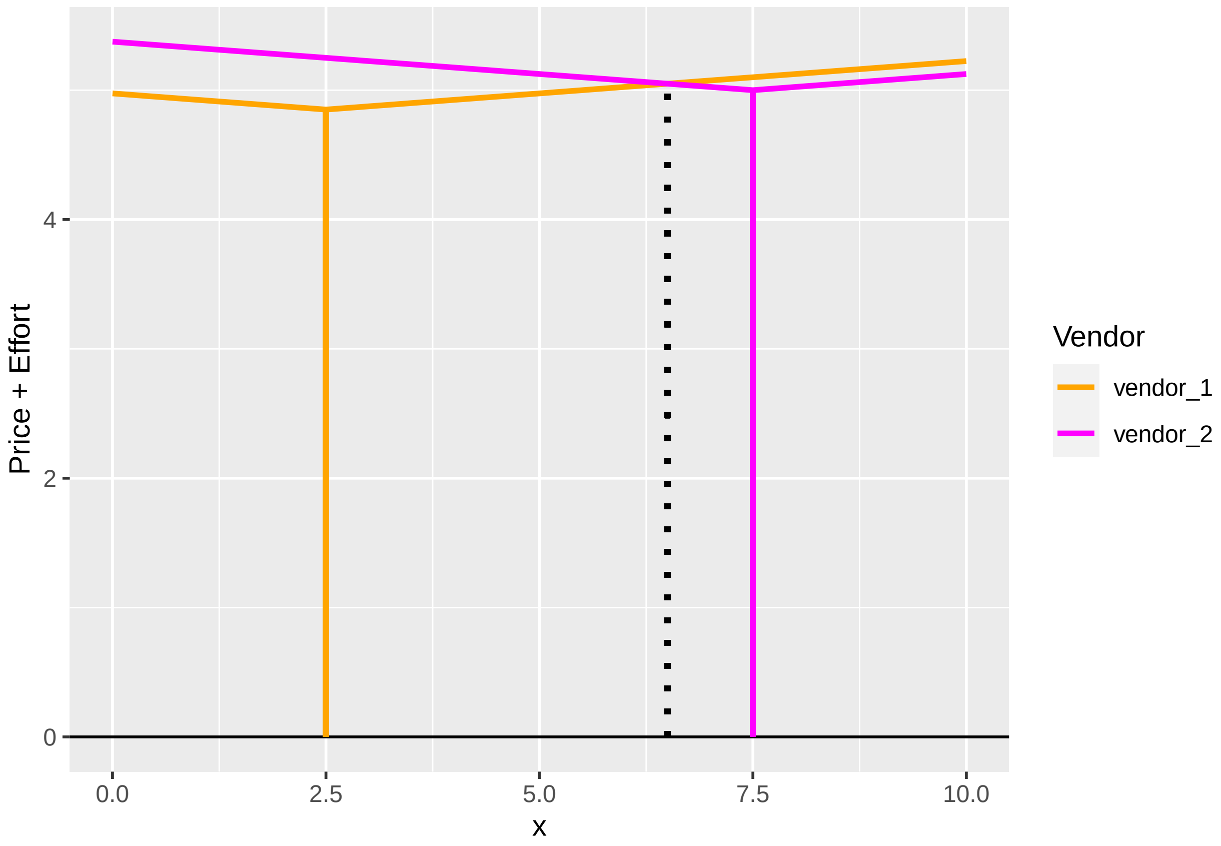
Another strategy that vendor 1 could use to increase their share of the market (instead of relocating, which might lose them some customers at the end of the beach) is to offer the same ice-cream (the quality is not compromised), but at a lower price. We will return the vendors to their original positions, and change the price instead:

```
location_1 = 2.5 # Location of vendor 1 (this must always be the one on the left)
location_2 = 7.5 # Location of vendor 2 (this must always be the one on the right)
price_1 = 4.85 # Price of ice-cream of vendor 1
price_2 = 5 # Price of ice-cream of vendor 2
delivery_1 = 0.05 # This is the effort needed to reach ice-cream vendor 1 (on the left)
delivery_2 = 0.05 # This is the effort needed to reach ice-cream vendor 2 (on the right)
```

Vendor 1 is back to location 2.5, but now he sells ice-cream for $4.85 (15 cents less expensive than the magenta ice-cream cart). Recreate the data frame with these new conditions, and plot the market areas:

```
market_areas <- data.frame(x = seq(from = 0, to = 10, by = 0.1),
                           price_1 = price_1,
                           price_2 = price_2,
                           delivery_1 = delivery_1,
                           delivery_2 = delivery_2,
                           location_1 = location_1,
                           location_2 = location_2) %>%
  mutate(vendor_1 = price_1 + abs(location_1 - x) * delivery_1, # Market area 1 depends on the location
         vendor_2 = price_2 + abs(location_2 - x) * delivery_2) # Market area 2 depends on the location

market_areas %>% # Pass market areas to the next function
  select(x, # Select three variables from the data frame: `x`, `vendor_1`, and `vendor_2`
         vendor_1,
         vendor_2) %>%
  pivot_longer(cols = starts_with("vendor"), # Pivot longer so that each row is point in space and the
               names_to = "Vendor",
               values_to = "Price + Effort") %>%
  mutate(Market = factor(Vendor, # Convert the variable `Vendor` to a factor with labels `Vendor 1` and
                         levels = c("vendor_1", "vendor_2"),
                         labels = c("Vendor 1", "Vendor 2"))) %>%
  ggplot(aes(x = x, y = `Price + Effort`, color = Vendor)) + # Create a ggplot object; the x-axis is th
  geom_line(size = 1) + # Plot the total price + effort as lines
  geom_hline(yintercept = 0,  # Plot a horizontal line: this is the beach
             color = "black") +
  geom_segment(x = location_1, # Plot a segment to indicate the price of the ice-cream offered by vendo
               xend = location_1,
               y = 0,
               yend = price_1,
               color = "orange", # The color of this line is orange
               size = 1) +
  geom_segment(x = location_2, # Plot a segment to indicate the price of the ice-cream offered by vendo
               xend = location_2,
               y = 0,
               yend = price_2,
               color = "magenta", # The color of this line is magenta
               size = 1) +
  geom_segment(x = (price_2 - price_1 + location_1 * delivery_1 + location_2 * delivery_2)/(delivery_1 +
               xend = (price_2 - price_1 + location_1 * delivery_1 + location_2 * delivery_2)/(delivery_
               y = 0,
               yend = price_2 + (location_2 - (price_2 - price_1 + location_1 * delivery_1 + location_2
               color = "black",
               linetype = "dotted",
               size = 1) +
  scale_color_manual(values = c("vendor_1" = "orange", "vendor_2" = "magenta")) # Set the colors of the
```



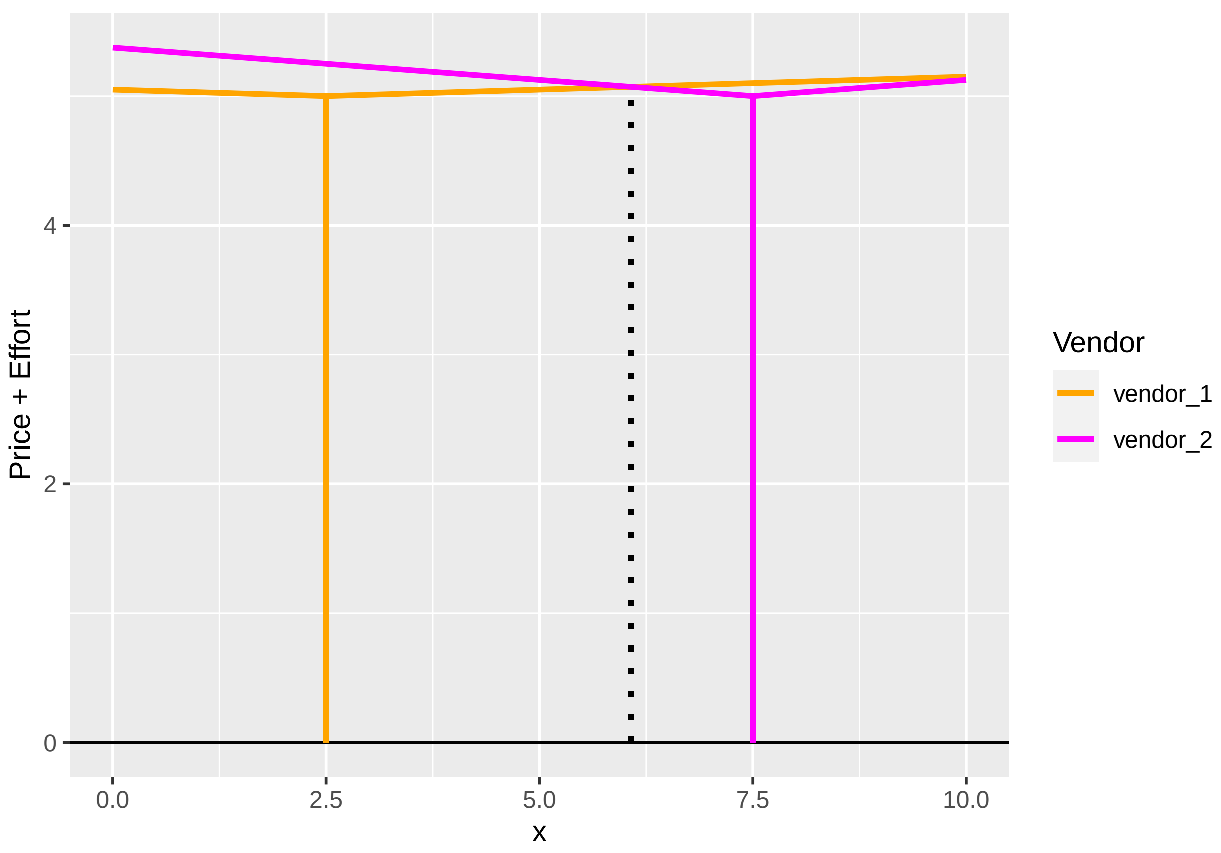
To point of indifference has been pushed away to the advantage of the orange cart, who now has a bigger share of the market. Finally, we will suppose that the orange vendor *cannot* reduce the price of his ice-cream without losing money…but they have discovered that being closer to the water makes a walk more comfortable than walking on the hot sand away from the water. By staying in the same point along the beach, but moving closer to the water, this vendor can reduce the effort that beach-goers need to reach him. We will simulate this by reducing the value of `deliver_1`:

```
location_1 = 2.5 # Location of vendor 1 (this must always be the one on the left)
location_2 = 7.5 # Location of vendor 2 (this must always be the one on the right)
price_1 = 5 # Price of ice-cream of vendor 1
price_2 = 5 # Price of ice-cream of vendor 2
delivery_1 = 0.02 # This is the effort needed to reach ice-cream vendor 1 (on the left)
delivery_2 = 0.05 # This is the effort needed to reach ice-cream vendor 2 (on the right)
```

We can plot the market areas under this new situation:

```
market_areas <- data.frame(x = seq(from = 0, to = 10, by = 0.1),
                           price_1 = price_1,
                           price_2 = price_2,
                           delivery_1 = delivery_1,
                           delivery_2 = delivery_2,
                           location_1 = location_1,
                           location_2 = location_2) %>%
  mutate(vendor_1 = price_1 + abs(location_1 - x) * delivery_1, # Market area 1 depends on the location
         vendor_2 = price_2 + abs(location_2 - x) * delivery_2) # Market area 2 depends on the location

market_areas %>% # Pass market areas to the next function
  select(x, # Select three variables from the data frame: `x`, `vendor_1`, and `vendor_2`
         vendor_1,
         vendor_2) %>%
  pivot_longer(cols = starts_with("vendor"), # Pivot longer so that each row is point in space and the
               names_to = "Vendor",
               values_to = "Price + Effort") %>%
  mutate(Market = factor(Vendor, # Convert the variable `Vendor` to a factor with labels `Vendor 1` and
                         levels = c("vendor_1", "vendor_2"),
                         labels = c("Vendor 1", "Vendor 2"))) %>%
  ggplot(aes(x = x, y = `Price + Effort`, color = Vendor)) + # Create a ggplot object; the x-axis is th
  geom_line(size = 1) + # Plot the total price + effort as lines
  geom_hline(yintercept = 0,  # Plot a horizontal line: this is the beach
             color = "black") +
  geom_segment(x = location_1, # Plot a segment to indicate the price of the ice-cream offered by vendo
               xend = location_1,
               y = 0,
               yend = price_1,
               color = "orange", # The color of this line is orange
               size = 1) +
  geom_segment(x = location_2, # Plot a segment to indicate the price of the ice-cream offered by vendo
               xend = location_2,
               y = 0,
               yend = price_2,
               color = "magenta", # The color of this line is magenta
               size = 1) +
  geom_segment(x = (price_2 - price_1 + location_1 * delivery_1 + location_2 * delivery_2)/(delivery_1 +
               xend = (price_2 - price_1 + location_1 * delivery_1 + location_2 * delivery_2)/(delivery_
               y = 0,
               yend = price_2 + (location_2 - (price_2 - price_1 + location_1 * delivery_1 + location_2
               color = "black",
               linetype = "dotted",
               size = 1) +
  scale_color_manual(values = c("vendor_1" = "orange", "vendor_2" = "magenta")) # Set the colors of the
```



Again, we see that the point of indifference has been pushed by this "innovation" that reduces the cost (i.e., the effort) of movement.

In real life, businesses use a mix of strategies with the purpose of increasing their margin of profits: they can sell poor quality products at very low prices, or very good products at reasonable prices. They can sacrifice some profits at the level of each sale, but grow their market areas and increase their revenue by having more clients. The bottom line is that there is an incentive for firms to be efficient, and that one way of achieving cost savings is by reducing how much they (or their customers) need to spend on transportation.

Which brings us to the transportation problem.

Preliminaries

As usual, before starting, it is useful to make sure that you have a clean Environment (unless there are values that you want to keep for further use). The following command will remove all data and values currently in the Environment:

```
rm(list = ls())
```

Load the data object with the road network:

```
data("hamilton_graph")
```

After loading the data file will can inspect the data object `hamilton_graph`:

```
hamilton_graph
```

```
## # A tbl_graph: 25816 nodes and 35142 edges
## #
## # An undirected multigraph with 47 components
## #
## # Edge Data: 35,142 x 6 (active)
##    from    to highway                                    geometry edgeID length
##   <int> <int> <chr>                                <LINESTRING [m]>  <int>    [m]
## 1     1     2 motorway      (570856.5 4811539, 570733.1 4811458, ~      1   224.
## 2     3     4 motorway_link (566453 4811377, 566503.5 4811218, 56~      2   310.
## 3     5     6 motorway_link (570893 4811386, 570880.6 4811374, 57~      3   307.
## 4     7     8 motorway_link (570487.2 4811279, 570523.1 4811291, ~      4   325.
## 5     1     9 motorway_link (570745.8 4811597, 570757.4 4811614, ~      5   363.
## 6    10    11 motorway_link (570812.7 4811680, 570795.7 4811671, ~      6   110.
## # ... with 35,136 more rows
## #
## # Node Data: 25,816 x 3
##   nodeID           geometry GTA06
##    <int>        <POINT [m]> <chr>
## 1      1 (570856.5 4811539) <NA>
## 2      2 (570667.7 4811419) <NA>
## 3      3   (566453 4811377) <NA>
## # ... with 25,813 more rows
```

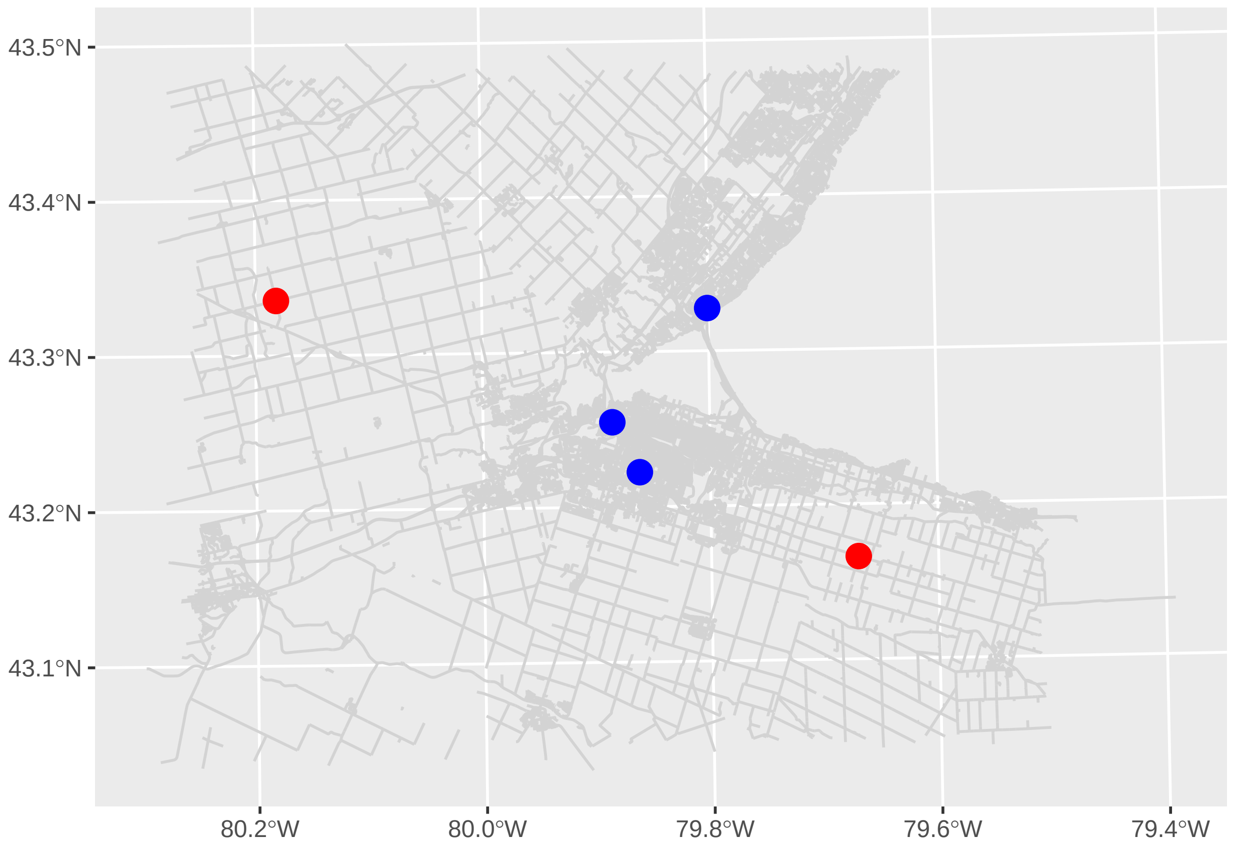
This is the road network that you worked with in Exercise 4. Check the notes for that exercise if you need a refresher about this object.

Motivating the transportation problem

1156 15719 17436 19260 25290

Consider a business with two small factories and three retail locations in the metropolitan region of Hamilton. The locations of the factories are approximated by nodes 25290 and 19260 in the network. The locations of the stores are approximated by nodes 1156, 15719, and 17436 in the network. The locations of factories and stores can be seen in the map below:

```
ggplot() + # Create a blank ggplot object
  geom_sf(data = hamilton_graph %>% # Pass the tbl_graph object to the next function to activate the ed
            activate(edges) %>% # Activate the edges of object `hamilton_graph_zoned`
            as_tibble() %>% # Convert to table
            st_as_sf(), # Convert to simple features for plotting the geo-spatial information
          color = "lightgray") + # Set the color of the edges to light gray
  geom_sf(data = hamilton_graph %>% # Pass the tbl_graph object to the next function to activate the no
            activate(nodes) %>% # Activate the nodes of object `hamilton_graph_zoned`
            as_tibble() %>% # Convert to table
            st_as_sf() %>% # Convert to simple features for plotting the geo-spatial information
            filter(nodeID == 25290 | nodeID == 19260), # Filter the nodes that correspond to the two fa
          color = "red", # Plot the factory nodes in red
          size = 4)  + # Set the size of the points
  geom_sf(data = hamilton_graph %>% # Pass the tbl_graph object to the next function to activate the no
            activate(nodes) %>% # Activate the nodes of object `hamilton_graph_zoned`
            as_tibble() %>% # Convert to table
            st_as_sf() %>% # Convert to simple features for plotting the geo-spatial information
            filter(nodeID == 1156 | nodeID == 15719 | nodeID == 17436), # Filter the nodes that corresp
          color = "blue", # Plot the factory nodes in blue
          size = 4) # Set the size of the points
```



The two factories are in the periphery of the metropolitan region, Factory I to the west, and Factory II to the east. The three stores are centrally located, one in Burlington (Store A), one in downtown Hamilton (Store B), and the last in the Mountain (Store C).

Factory I can produce 800 units of candy per quarter, whereas the capacity of Factory II is 600 units of candy per quarter. Quarterly demand of candy at each of the stores is estimated at 500 units in Burlington, 400 units in Hamilton, and 400 units in the Mountain. This means that there are six possible flows, from Factory I to Stores A, B, and C (call these flows x_{11}, x_{12}, x_{13}), and from Factory II to Stores A, B, and C (call these flows x_{21}, x_{22}, x_{23}). The cost of transporting the product from the factories to the stores is known. The cost depends on the distance between the factory and the store, as well as other factors, such as congestion, possible delays, the physical configuration of the stores (is there a dock to speed unloading), and so on.

The flows, on the other hand, are our decision variables: we need to decide a delivery schedule, or in other words, how many units of candy to ship from each factory to each store.

This information can be collected in the form of an origin-destination matrix and a cost matrix.

Origin-destination matrix:

Factory	Store A	Store B	Store C	Production
Factory I	x_{11}	x_{12}	x_{13}	800
Factory II	x_{21}	x_{22}	x_{23}	600
Demand	500	400	400	

Cost matrix:

Factory	Store A	Store B	Store C
Factory I	16	20	22

Factory	Store A	Store B	Store C
Factory II	18	16	14

We can write the origin destination matrix in this way:

```
od_matrix <- c(x11, x12, x13, x21, x22, x23) %>% # Take this vector of values and pass to the next func
  matrix(nrow = 2, byrow = TRUE) # Convert to a matrix with two rows; the data are organized by rows
```

And the cost matrix:

```
cost_matrix <- c(16, 20, 22, 18, 16, 14) %>% # Take this vector of values and pass to the next function
  matrix(nrow = 2, byrow = TRUE) # Convert to a matrix with two rows; the data are organized by rows
cost_matrix
```

```
##      [,1] [,2] [,3]
## [1,]   16   20   22
## [2,]   18   16   14
```

If you run the chunk that creates the origin-destination matrix it will give you an error: the computer does not know what `x11` is (or for that matter `x12`, `x12`, etc.) It does not know, because we have not told it yet what those values are, because we have not made a decision. The problem is that, even in this small example, there are many different ways of distributing the candy to the stores using six routes. For instance, suppose that we make the following decision:

```
# Assign values to the decision variables
x11 <- 0 # Send this number of units from Factory I to Store A
x12 <- 400 # Send this number of units from Factory I to Store B
x13 <- 400 # Send this number of units from Factory I to Store C
x21 <- 500 # Send this number of units from Factory II to Store A
x22 <- 0 # Send this number of units from Factory II to Store B
x23 <- 0 # Send this number of units from Factory II to Store C

od_matrix <- c(x11, x12, x13, x21, x22, x23) %>% # Take this vector of values and pass to the next func
  matrix(nrow = 2, byrow = TRUE) # Convert to a matrix with two rows; the data are organized by rows
od_matrix
```

```
##      [,1] [,2] [,3]
## [1,]    0  400  400
## [2,]  500    0    0
```

Here, our decision is to send 400 units of candy from Factory I to Store B, 400 units from Factory I to Store C, and 500 units from Factory II to Store A. Is this delivery schedule reasonable? And by reasonable we set a relatively low bar: we cannot ship more candy than we have at the factories, and we want to deliver at least as much as the stores need. We can check whether these conditions are met

Verify that the flows do not exceed the supply:

```
check_condition <- ifelse(x11 + x12 + x13 <= 800, # The sum of flows from Factory I is less than or equ
                          c("Supply from Warehouse I is sufficient"), # If true
                          c("Supply from Warehouse I exceeded")) # If false

print(check_condition) # Print message
```

```
## [1] "Supply from Warehouse I is sufficient"
```

```
check_condition <- ifelse(x21 + x22 + x23 <= 600, # The sum of flows from Factory I is less than or equ
                          c("Supply from Warehouse II is sufficient"), # If true
                          c("Supply from Warehouse II exceeded")) # If false

print(check_condition) # Print message
```

```
## [1] "Supply from Warehouse II is sufficient"
```

Verify that the flows satisfy demand:

```
check_condition <- ifelse(x11 + x21 >= 500, # The flows to Store A are greater than or equal to 500
                          c("Demand at store A is met"), # If true
                          c("Demand at store A is not met")) # If false

print(check_condition) # Print message
```

```
## [1] "Demand at store A is met"
```

```
check_condition <- ifelse(x12 + x22 >= 400, # The flows to Store A are greater than or equal to 400
                          c("Demand at store B is met"), # If true
                          c("Demand at store B is not met")) # If false

print(check_condition) # Print message
```

```
## [1] "Demand at store B is met"
```

```
check_condition <- ifelse(x13 + x23 >= 400, # The flows to Store A are greater than or equal to 400
                          c("Demand at store C is met"), # If true
                          c("Demand at store C is not met")) # If false

print(check_condition) # Print message
```

```
## [1] "Demand at store C is met"
```

The distribution schedule is reasonable in that it does not ask factories to send more than they can, and all stores receive at least as much as they need. But is it efficient? The cost of distribution by route according to this schedule is:

```
od_matrix * cost_matrix # Multiply the number of units per route by the unit transportation cost
```

```
##      [,1] [,2] [,3]
## [1,]    0 8000 8800
## [2,] 9000    0    0
```

And the total cost is the sum of the cost by route:

```
sum(od_matrix * cost_matrix)
```

```
## [1] 25800
```

The cost of this distribution schedule is 25,800 (cents). We can try a different schedule:

```
# Assign values to the decision variables
x11 <- 500 # Send this number of units from Factory I to Store A
x12 <- 300 # Send this number of units from Factory I to Store B
x13 <- 0 # Send this number of units from Factory I to Store C
x21 <- 0 # Send this number of units from Factory II to Store A
x22 <- 100 # Send this number of units from Factory II to Store B
x23 <- 400 # Send this number of units from Factory II to Store C

od_matrix <- c(x11, x12, x13, x21, x22, x23) %>% # Take this vector of values and pass to the next func
  matrix(nrow = 2, byrow = TRUE) # Convert to a matrix with two rows; the data are organized by rows
od_matrix
```

```
##      [,1] [,2] [,3]
## [1,]  500  300    0
## [2,]    0  100  400
```

Here, our decision is to send 500 units of candy from Factory I to Store A, 300 units from Factory I to Store B, 100 units from Factory II to Store B, and 300 units from Factory II to Store C. Verify that the flows do not exceed the supply:

```
check_condition <- ifelse(x11 + x12 + x13 <= 800, # The sum of flows from Factory I is less than or equ
                          c("Supply from Warehouse I is sufficient"), # If true
                          c("Supply from Warehouse I exceeded")) # If false

print(check_condition) # Print message
```

```
## [1] "Supply from Warehouse I is sufficient"
```

```
check_condition <- ifelse(x21 + x22 + x23 <= 600, # The sum of flows from Factory I is less than or equ
                          c("Supply from Warehouse II is sufficient"), # If true
                          c("Supply from Warehouse II exceeded")) # If false

print(check_condition) # Print message
```

```
## [1] "Supply from Warehouse II is sufficient"
```

Verify that the flows satisfy demand:

```
check_condition <- ifelse(x11 + x21 >= 500, # The flows to Store A are greater than or equal to 500
                          c("Demand at store A is met"), # If true
                          c("Demand at store A is not met")) # If false

print(check_condition) # Print message
```

```
## [1] "Demand at store A is met"
```

```
check_condition <- ifelse(x12 + x22 >= 400, # The flows to Store A are greater than or equal to 400
                          c("Demand at store B is met"), # If true
                          c("Demand at store B is not met")) # If false

print(check_condition) # Print message
```

```
## [1] "Demand at store B is met"
```

```
check_condition <- ifelse(x13 + x23 >= 400, # The flows to Store A are greater than or equal to 400
                          c("Demand at store C is met"), # If true
                          c("Demand at store C is not met")) # If false

print(check_condition) # Print message
```

```
## [1] "Demand at store C is met"
```

Again, the distribution schedule is reasonable. However, the total cost of distribution according to this schedule is:

```
sum(od_matrix * cost_matrix)
```

```
## [1] 21200
```

Which is cheaper than the previous one. Clearly, there are better and worse ways of designing a distribution schedule, and our problem is to find an efficient one, hopefully even the *most* efficient one. In a small problem like this example, it is possible to find the most efficient distribution schedule by trial and error (you can try it if you want), but this is just not feasible for most real world logistics networks.

What we need instead, is a procedure (i.e., an algorithm) that can solve the transportation problem in a systematic, and preferably automated (or at least semi-automated) way.

Control flow

Speaking of automation, it is convenient at this point to introduce an important concept in working with algorithms. The concept is *control flow*. Being able to choose which instructions to execute is a powerful way of controlling the flow of a set of instructions, procedure, or algorithm. Many computer languages include structures to achieve just that.

if-(then)-else statements

An important control flow structure is if-(then)-else statatments. We have already seen statements of this kind before, most recently in line 510 of this notebook above. An if-(then)-else statement will execute a set of instructions *if* a logical statement evaluates to `TRUE`. Otherwise, it will skip the instructions and proceed with the rest of the code.

In addition to the `ifelse()` function that we use above, `R` can handle if-(then)-else statements like this:

```
x <- 5 # Assign a value to variable `x`
if(x > 0){ # `x` is greater than zero
  print("x is a positive number") # If true, print this message
}
```

```
## [1] "x is a positive number"
```

The `if` condition is evaluated (in this case to see whether the value of `x` is greater than zero). If true *then* the set of instructions between the curly brackets is executed. If false, then the instructions are ignored. Run the chunk. Try changing the value of `x` to see what happens.

An `if` statement can also include an explicit set of `else` instructions, that is, things to do in case the statement is false. See for example:

```
x <- 5
if(x > 0){
  print("x is a positive number")
}else{
  print("x is a negative number")
}
```

```
## [1] "x is a positive number"
```

Again, try setting different values of `x` in the code chunk above to see what happens.

If you try `x <- 0` you will notice that the logic is somewhat flawed, because 0 is neither positive nor negative. To fix this, we can use nested `if` statements, to ensure that the no outcomes are overlooked. See:

```
x <- 5
if(x > 0){
  print("x is a positive number")
}else if(x < 0){
  print("x is a negative number")
}else{
  print("x is zero")
}
```

```
## [1] "x is a positive number"
```

If `x` is a positive number the first `print()` instruction will be executed, and the rest will be ignored. If `x` is a negative number, the first `print()` instruction will be ignored, the second condition will evaluate as `TRUE` (and therefore the second `print()` will be executed), and the last instruction will be ignored. If `x` was not positive (first condition) and not negative (second condition), then `x` *must* be zero, because numbers are positive, negative, or zero!

This ignores other possible situations (for instance, that `x` is not a number but instead, say, a letter). Exception handling is an important part of coding, but to keep this simple we will assume that the inputs are of the correct type.

Loops

Loops are a somewhat different category of conditional branching. Loops execute a set of instructions zero, one, or more times, depending on a condition.

Three loop structures in `R` are as follows:

- `for`: a `for` statement will execute the set of instructions a predetermined number of times.
- `while`: a `while` statement will execute the set of instructions an indeterminate number of times, while a certain condition is met.
- `repeat`: this is similar to while, but the condition is evaluated at the end of the set of instructions, instead of at the beginning as in `while`. This ensures that the instructions are executed *at least once*.

We will next illustrate these control flow structures, beginning with `for`.

```
for(i in 1:10){
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

The loop above executes one single instruction (to print `i`) a determined number of times (from 1 to 10, as seen in the `for` statement). We can easily modify this so that the loop reports the square of `i` every time. Notice how we can use `i` as a *counter*, to keep track of where we are in the loop, and also as a variable. For example:

```r
for(i in 1:10){
  x <- i^2
  print(x)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
## [1] 36
## [1] 49
## [1] 64
## [1] 81
## [1] 100
```

The loop above reported the squares of all numbers from 1 to 10. We can also modify how `i` behaves, by using a sequence of numbers. Whereas in the previous chunks the sequence `1:10` was all numbers from 1 to 10, the sequence `c(2,4,6,8,10)` includes only the even integers between 1 and 10:

```r
for(i in c(2,4,6,8,10)){
  x <- i^2
  print(x)
}
```

```
## [1] 4
## [1] 16
## [1] 36
## [1] 64
## [1] 100
```

Notice that the instructions contained in the loop will be executed *exactly* the numbers of times predefined in the sequence provided in the statement.

The `while` loop works differently. Sometimes we do not know before hand how long it will take to complete a task, so we wish the instructions to run as long as necessary until some predefined condition is met. This requires us to initialize said condition before entering the `while` loop. We can illustrate this using the `sample()` function to generate random integers, with or without replacement. Basically, the loop in the chunk below will print the value of `x` before substracting from it a random number between 1 and 3, until `x` becomes zero or a negative value:

```r
x <- 10 #initialize x
while(x > 0){
  print(x)
  z <- sample(1:3, 1, replace = TRUE)
  x <- x - z
}
```

```
## [1] 10
## [1] 8
## [1] 6
## [1] 4
## [1] 1
```

If you assign zero or a negative value to `x` at the moment of initialization, the instructions in the loop will not be executed. Try it. Since we do not know how many *iterations* (repetitions of the instructions) it will take for `x` to become zero or negative (there is an element of randomness to it), we cannot use a `for` loop in this situation.

The structure of `repeat` is as follows, with the exit condition coming at the end in the form of the `break` instruction. Notice that even if `x` is initialized as zero, the set of instructions will be executed at least once:

```
x <- 10
repeat {
  print(x)
  z <- sample(1:3, 1, replace = TRUE)
  x <- x - z
  if (x < 0){
    break
  }
}
```

```
## [1] 10
## [1] 7
## [1] 4
## [1] 3
## [1] 1
```

Loops are important elements in coding, because they allow us to write sets of instructions in a compact way. The transportation problem of operations research is an excellent example of this, because similar operations are repeated until a certain condition is achieved. This will be illustrated next.

The transportation problem defined

Let us return to the example above, with two factories (I and II), and three stores (A, B, and C).

The primal problem

In the examples above we calculated the total cost of distributions as the sum of the products of the flows by their unit cost, according to the route used to ship them. More formally, we can write the total cost of distribution in this way:

$$C = 16x_{11} + 20x_{12} + 22x_{13} + 18x_{21} + 16x_{22} + 14x_{23}$$

This is our *objective function*, the quantity that we use as our criterion to decide how efficient our decisions are. Previously, a decision with values $x_{12} = 400$, $x_{13} = 400$, and $x_{21} = 500$ was not as efficient as a decision with values $x_{11} = 500$, $x_{12} = 300$, $x_{22} = 100$, and $x_{23} = 400$, which achieved our goal of satisfying the demand without exceeding the amount of candy the factories could supply, and at a lower cost too. So, in reality, our objective is to find the decision variables that *minimize* the cost:

$$minC = 16x_{11} + 20x_{12} + 22x_{13} + 18x_{21} + 16x_{22} + 14x_{23}$$

The conditions that the factories have a limit to how much candy they can supply are written as follows:

$$\begin{aligned} x_{11} + x_{12} + x_{13} \leq 800 \\ x_{21} + x_{22} + x_{23} \leq 600 \end{aligned}$$

We call these the *supply constraints*. The conditions that stores must receive at least as much as they need are the *demand constraints*, and are written like so:

$$x_{11} + x_{21} \geq 500$$
$$x_{12} + x_{22} \geq 400$$
$$x_{13} + x_{23} \geq 400$$

In addition to suppy and demand constraints, we also need to define another set of constraints that are perhaps a little less obvious:

$$x_{11} \geq 0; x_{12} \geq 0; \cdots ; x_{23} \geq 0;$$

These constraints are called *non-negativity constraints*, and basically say that flows must move in one direction, from factory to store. Negative flows would move from store from factory, something wasteful that does not help us to achieve our goal of having candy in the stores. Notice that the supply constraints are inequalities of the form "less than or equal to" where the demand constraints are inequalities of the form "greater than or equal to". For technical reasons, it is convenient to have all inequalities of the same type, something that we can achieve by multiplying both sides of the supply constraints by minus one:

$$-x_{11} - x_{12} - x_{13} \geq -800$$
$$-x_{21} - x_{22} - x_{23} \geq -600$$

The objective function and the constraints can be written in the form of a matrix:

x_{11}	x_{12}	x_{13}	x_{21}	x_{22}	x_{23}	Constant
-1	-1	-1	0	0	0	-800
0	0	0	-1	-1	-1	-600
1	0	0	1	0	0	500
0	1	0	0	1	0	400
0	0	1	0	0	1	400
16	20	22	18	16	14	

If we examine the table we see that at the top we have the unknown flows (x_{11} through x_{23}). The bottom row is the objective function (the cost multiplied by the flows) and the rightmost column are the constraints. The pattern of ones and zeros in the table indicates the possible flows; for example, the first column says that an outgoing flow from Factory I to Store A (the negative one in the second row) must match an incoming flow to Store A from Factory 1 (the positive one in the third row). In a similar fashion, we we inspect the second row, it says that all outgoing flows from Factory I (the negative ones) are constrained by the total supply at the factory (the negative 800).

The dual problem

The objective function (basically the cost function), and the constraints, together constitute the *primal* version of the transportation problem (so called because it is the first formulation of the problem). The primal of the problem can be used to obtain the *dual* version of the problem. If we interchange the rows and columns of the matrix (this is called the *transpose* of the matrix), we obtain the following:

s_1	s_2	s_3	s_4	s_5	Constant
-1	0	1	0	0	16
-1	0	0	1	0	20
-1	0	0	0	1	22
0	-1	1	0	0	18
0	-1	0	1	0	16
0	-1	0	0	1	14

s_1	s_2	s_3	s_4	s_5	Constant
-800	-600	500	400	400	

This second matrix encodes the same information as the first one, but in a different way. Now, the variables in the first row of the table are not flows, but rather represent *stocks* at each of the five locations in our system: s_1 and s_2 are the columns for Factories I and II, respectively. Columns s_3, s_4, and s_5 are for Stores A, B, and C. The bottom row of this table says that we have 800 units and 600 units in Factories A and B that are "negative" in value (since candy at the factory cannot be sold), and that we have a demand of candy in Stores A, B, and C equal to 500, 400, and 400 respectively. The rightmost column now contains the unit cost of transportation.

Working from the transposed matrix, the dual problem can be formulated with the following objective function:

$$P = -800s_1 - 600s_2 + 500s_3 + 400s_4 + 400s_5$$

In the equation above, P is a made up quantity that is a summary of the value of the stock available at each site. We can see that stock that remains at the factories (s_1 and s_2) detracts from this quantity, whereas stock at the stores (s_3, s_4, and s_5) adds value. Whereas in the primal problem we wanted to *minimize* the cost, the dual problem offers a different perspective, where we wish to *maximize* the value of our stock:

$$maxP = -800s_1 - 600s_2 + 500s_3 + 400s_4 + 400s_5$$

This problem also has a set of constraints, as follows:

$$\begin{gathered}
-s_1 + s_3 \leq 16\\
-s_1 + s_4 \leq 16\\
-s_1 + s_5 \leq 16\\
-s_2 + s_3 \leq 16\\
-s_2 + s_4 \leq 16\\
-s_2 + s_5 \leq 16\\
s_1 \geq 0; \cdots ; s_5 \geq 0
\end{gathered}$$

These constraints establish that the cost of transferring a unit of stock from a factory to a store must be *at most* the cost on the route (we would not want to pay more than that!)

The simplex tableau

Both the primal and dual formulations are needed to solve the problem: they ensure that we are moving towards a solution from above (by minimizing the cost) and below (by maximizing the value of stock). This can be seen in the way the *simplex tableu* is constructed (simplex being the name of the algorithm used to solve the transportation problem).

Before we construct the tableau, we note that the objective function of the dual problem can be re-written as:

$$P + 800s_1 + 600s_2 - 500s_3 - 400s_4 - 400s_5 = 0$$

To construct the simplex tableau, we augment the matrix of the dual problem with an identity matrix (a matrix with ones in the diagonal and zeros elsewhere) as follows:

The last row of this table tells us (in the first column) that we have 800 units at Factory I that are available; in the second column, likewise, it says that we have 600 units there available to move; the next three columns say that we need 500 units in Store A, 400 in Store B, and 400 in Store C. The last row of the columns corresponding to x_{11} through x_{23} are all zeros because we have not allocated any flows yet.

We are now ready to solve the transportation problem.

The algorithm for the first step is as follows:

1. Identify the column with the *most* negative value at the bottom. This column becomes the *pivot column* (In our initial tableau, this is column 3, for s_3, the store with the largest amount of unmet demand).

2. In that column, find the ones (which tell us which Factories are available to ship candy). There are two ones in the initial tableu, in rows one and four, indicating the possibility of receiving candy from Factories I and II (also see the ones in the same rows under x_{11} and x_{21}). So we look at the cost associated with allocating those flows: the last column has a value of 16 in the first row, and a value of 18 in the fourth row. Since we want to minimize the cost of transportation, we choose the first row that has the lowest cost of the two routes available. The intersection of this row with the pivot column becomes the *pivot element*. (In our initial tableau, the one in the first row and third column).

3. Once we identified the site with the largest unmet demand and the flow with the lowest cost that can satisfy that demand, we need to update the tableu. This is done by using the pivot column into a *unit column*, that is, a column with a one in the position of the pivot element and zeros elsewhere.

To convert the pivot column into a unit column we use the pivot row to transform other values in the column to zeros. In the initial tableau we need to convert the one in the fourth row into a zero, and the -500 in the last row into a zero. So, we take row 1 (or R_1) and do the following:

$$R_4 - R_1$$
$$R_7 + 500R_1$$

Since the value in the pivot column in row 4 is one and in row 1 is one, the operation gives a zero. Likewise, since the value in the pivot column in row 7 is -500 and the value in row 1 is one, the operation also gives a zero. The same operation needs to be conducted along the two rows that we are updating! See below for row 4:

If we try and look beyond the mechanics of these operations, notice (see the bottom row) how the stock at Factory I has been reduced by 500, the demand at Store A is now zero, and we have a flow of 500 going between Factory I and Store A. Furthermore, the total cost of transportation so far is 8,000 (last column of bottom row); this is the result of multiplying 500 units moved by the unit cost in that route, which was 16. We also see that the cost in row 4 is now 2: this is how much more expensive that route is relative to the least expensive alternative. Once we have exhausted the least expensive alternatives, this value tells us what is the additional cost per unit of using this route.

We do not know at the outset how many steps it will take to solve the problem, but we can invoke a stopping criterion. We will say that *while* there are negative values in the bottom row we will continue with at least one more iteration. Negative values in the bottom row signify one of two things: unmet demand (as in the columns for s_4 and s_4 in the updated tableau), or requesting more than a source can provide (if we had negative values at the bottom of columns s_1 and/or s_2). While we have negative values in the bottom row, we will continue with the algorithm as follows:

1. Check for negative values in the bottom row. If true, then proceed to step 2; if false, stop.

2. Find the most negative value in the bottom row; this becomes the pivot column. In the case of ties, choose any of the tied columns.

3. Find all positive 1's in the pivot column; check the last column of the rows with positive 1's and find the minimum cost. The intersection of the row with the minimum cost and the pivot column becomes the pivot element.

4. Use the row with the pivot element to convert the pivot column into a unit column. This will update the tableau.

5. Return to step 1.

As we can see, this algorithm involves if-(then)-else statements and *while* loop. Next, we will see how to use flow control statements to automate the transportation problem.

Using control flow to solve the transportation problem

We will begin by creating the initial simplex tableau. This is a matrix that contains the information discussed above:

```
s1 <- c(-1, -1, -1, 0, 0, 0, 800)
s2 <- c(0, 0, 0, -1, -1, -1, 600)
s3 <- c(1, 0, 0, 1, 0, 0, -500)
s4 <- c(0, 1, 0, 0, 1, 0, -400)
s5 <- c(0, 0, 1, 0, 0, 1, -400)
x11 <- c(1, 0, 0, 0, 0, 0, 0)
x12 <- c(0, 1, 0, 0, 0, 0, 0)
x13 <- c(0, 0, 1, 0, 0, 0, 0)
x21 <- c(0, 0, 0, 1, 0, 0, 0)
x22 <- c(0, 0, 0, 0, 1, 0, 0)
x23 <- c(0, 0, 0, 0, 0, 1, 0)
P <- c(0, 0, 0, 0, 0, 0, 1)
Constant <- c(16, 20, 22, 18, 16, 14, 0)
tableau <- cbind(s1, s2, s3, s4, s5, x11, x12, x13, x21, x22, x23, P, Constant)
tableau
```

```
##       s1  s2   s3   s4   s5 x11 x12 x13 x21 x22 x23 P Constant
## [1,]  -1   0    1    0    0   1   0   0   0   0   0 0       16
## [2,]  -1   0    0    1    0   0   1   0   0   0   0 0       20
## [3,]  -1   0    0    0    1   0   0   1   0   0   0 0       22
## [4,]   0  -1    1    0    0   0   0   0   1   0   0 0       18
## [5,]   0  -1    0    1    0   0   0   0   0   1   0 0       16
## [6,]   0  -1    0    0    1   0   0   0   0   0   1 0       14
## [7,] 800 600 -500 -400 -400   0   0   0   0   0   0 1        0
```

You can verify that this is identical to our initial tableau. We also save two variables, with the number of rows and the columns in our tableau:

```
nr <- nrow(tableau) # number of rows in the tableau
nr
```

```
## [1] 7
```

```
nc <- ncol(tableau) # number of columns in the tableau
nc
```

```
## [1] 13
```

Once the simplex tableau is ready, we need to identify the site with the highest level of unsatisfied demand (in row 7). We do this by means of the function `which()`, which identifies the index of values that satisfy a certain condition, presently the minimum value in the last row of the tableau:

```
pivotCol <- which(tableau[nr,] == min(tableau[nr,]))
pivotCol
```

```
## s3
##  3
```

This index serves to identify the *pivot column* (it can be seen here that the pivot column here is 3, corresponding to s3).

After identifying the pivot column, we must select the *pivot row*. The pivot row must be a row that contains a one in the pivot column. We can again use `which`, to find the index of values in the pivot column that are equal to 1. We call these `candidateRows`:

```
candidateRows <- which(tableau[,pivotCol] == 1)
candidateRows
```

```
## [1] 1 4
```

As seen above, there are two ones in the pivot column, in rows 1 and 4. To select the pivot row we need to look up the values in the last column (`Constant`). These values correspond to the unit transportation costs, and therefore we wish to select the one that represents the least cost. Again, using `which`, we look up the values of the rows with ones in the pivot column, and identify the lowest value in the last column:

```
pivotRow <- candidateRows[which(tableau[candidateRows, nc] == min(tableau[candidateRows, nc]))]
pivotRow
```

```
## [1] 1
```

```
tableau[pivotRow, nc]
```

```
## Constant
##       16
```

We have now identified the pivot column as the third column (`s3`), and the pivot row as the first row, with a constant (i.e., cost) of 16.

The next step is to convert the pivot column into a *unit column*, by transforming all non-zero values in that column into zeros. The basis for the calculations to follow is the pivot row. To proceed, we must identify all non-zero elements in the pivot column. We did this before by visual inspection (we looked over the simplex tableau and saw that the non-zero elements of the pivot column were in rows 4 and 7). Here, we use `which` again (note the use of `!=` for *different from*):

```
nonZero <- which(tableau[, pivotCol] != 0)
nonZero
```

```
## [1] 1 4 7
```

So, besides the pivot row (the first row, which we can ignore) there are two non-zero values in the pivot column: rows 4 and 7. We will remove the pivot row from the list of rows to update:

```
nonZero <- nonZero[which(nonZero != pivotRow)]
nonZero
```

```
## [1] 4 7
```

To update the table we need to convert the pivot column into a unit column. Take first the fourth row in the simplex tableau. Its non-zero value in the pivot column is one, and we wish to convert it into a zero. To do this, we can subtract the pivot row from row 4, after multiplying the pivot row by the pivot column element in row 4:

```
tableau[nonZero[1],] - tableau[nonZero[1], pivotCol] * tableau[pivotRow,]
```

```
##       s1       s2       s3       s4       s5      x11      x12      x13 
##        1       -1        0        0        0       -1        0        0 
##      x21      x22      x23        P Constant 
##        1        0        0        0        2
```

Notice how the value in the third column of row 4 now has become zero, as desired. The constant has become 2 (from 18). This is the marginal cost of movement when shipping from location I (i.e., 18 - 16). Now we can work on the seventh row in the simplex tableau. The pivot column element in column 3 is the value that we wish to convert to a zero, which in this case is -500. To convert this into a zero, we subtract the pivot row from row 7, after multiplying the pivot row by the -500:

```
tableau[nonZero[2],] - tableau[nonZero[2], pivotCol] * tableau[pivotRow,]
```

```
##       s1       s2       s3       s4       s5      x11      x12      x13 
##      300      600        0     -400     -400      500        0        0 
##      x21      x22      x23        P Constant 
##        0        0        0        1     8000
```

See how now the value in the third column (u3) has become zero, the value in the sixth column (x11) has become 500, and the constant has become 8,000. In other words, this operation has satisfied the supply in location u3, by assigning a flow of 500 to x11. The cost of doing this is 8,000 (you can verify, since it is the number of units moved, 500, times the unit cost, 16):

```
500 * 16
```

```
## [1] 8000
```

We just completed the first iteration of the simplex procedure. We could continue repeating these operations as appropriate to solve the transportation problem, but the operations are repetitive and tedious. Moreover, you've learned some simple rules at the start of this lesson that allow us to implement operations using conditional branching structures.

The rows with non-zero values in the pivot column include row number one, which is actually the pivot row. As noted above, we need to convert all other values to zeros, but not the pivot row. Let us sketch a loop that can do this:

```
for(i in 1:length(nonZero)){
  print(nonZero[i])
}
```

```
## [1] 4
## [1] 7
```

The loop simply prints the numbers of the rows that need to be updated to obtain the unit column. Now that we have sketched the loop, we can implement the instructions that we need to convert the pivot column into a unit column.

```
for(i in 1:length(nonZero)){
  tableau[nonZero[i],] <- tableau[nonZero[i],] - tableau[nonZero[i], pivotCol] * tableau[pivotRow,]
}
tableau
```

```
##       s1  s2 s3   s4   s5 x11 x12 x13 x21 x22 x23 P Constant
## [1,]  -1   0  1    0    0   1   0   0   0   0   0 0       16
## [2,]  -1   0  0    1    0   0   1   0   0   0   0 0       20
## [3,]  -1   0  0    0    1   0   0   1   0   0   0 0       22
## [4,]   1  -1  0    0    0  -1   0   0   1   0   0 0        2
## [5,]   0  -1  0    1    0   0   0   0   0   1   0 0       16
## [6,]   0  -1  0    0    1   0   0   0   0   0   1 0       14
## [7,] 300 600  0 -400 -400 500   0   0   0   0   0 1     8000
```

You can verify that this loop has successfully created a unit column to replace the pivot column (i.e., s_3). This is our new simplex tableau. We return to the step where we identify a pivot column. Notice, however, that there are two columns with identical levels of unsatisfied demand. Our protocol says that the pivot column must be the one with the most negative value in the last column. In the case of ties, we can select either column and the results are not affected. Computationally, however, we need to modify our procedure a little bit for the following reason:

```
pivotCol <- which(tableau[nr,] == min(tableau[nr,]))
pivotCol
```

```
## s4 s5
##  4  5
```

Our instruction now returns two values! This would make things confusing for the instructions, that have been designed for only *one* pivot column. Fortunately, since we can choose either column when there are ties, we can simply select the first of several candidates for pivot column, as follows:

```
pivotCol <- which(tableau[nr,] == min(tableau[nr,]))
pivotCol <- pivotCol[1]
pivotCol
```

```
## s4
##  4
```

Now we can execute the instructions unambiguously on column 4. These would include identifying the candidates for pivot row (in this case rows 2 and 5), then selecting a pivot row, then using the pivot row to convert the pivot column into a unit column. And so on.

As you can see, this is all a repetition of the same set of instructions. Instead of doing this all by hand, we can create a loop and save ourselves time and effort! Unlike the loop we used for creating the unit column which had to be executed a predefined number of times (once for each non-zero value in the pivot column besides the pivot row), when repeating these instructions we do not know at the outset how many times we need to repeat them. We do know, however, the condition to stop: the stopping criterion is when there are no negative values left in the last row, since all demand has been satisfied at all sites and we have not exceeded the supply at any site.

Let us try this. Given a simplex tableau, initialize the highest level of unsatisfied demand. This will be our stopping criterion. When this value is no longer negative, we can stop the algorithm:

```
stopCriterion <- min(tableau[nr,])
stopCriterion
```

```
## [1] -400
```

So now, instead of all of the above code, we can collect it all into an algorithm, and execute it together without all the superfluous printing of answers:

```
stopCriterion <- min(tableau[nr,])
while(stopCriterion < 0){ # While the stopping criterion is still negative, do the following instructio
  pivotCol <- which(tableau[nr,] == min(tableau[nr,])) # Identify candidates for pivot column
  pivotCol <- pivotCol[1] # Break possible ties in candidates for pivot column
  candidateRows <- which(tableau[,pivotCol] == 1) # Find candidate rows for pivot row
  pivotRow <- candidateRows[which(tableau[candidateRows, nc] == min(tableau[candidateRows, nc]))] # Ide
  nonZero <- which(tableau[, pivotCol] != 0) # Find non-zero elements in the pivot column
  nonZero <- nonZero[which(nonZero != pivotRow)] # Remove the pivot row from the list of rows with non-
  for(i in 1:length(nonZero)){ # For all rows that need to be updated, do the following instruction
    tableau[nonZero[i],] <- tableau[nonZero[i],] - tableau[nonZero[i], pivotCol] * tableau[pivotRow,] #
  }
  stopCriterion <- min(tableau[nr,]) # Update the stopping criterion
}
tableau
```

```
##       s1 s2 s3 s4 s5 x11 x12 x13 x21 x22 x23 P Constant
## [1,]  -1  0  1  0  0   1   0   0   0   0   0 0       16
## [2,]  -1  1  0  0  0   0   1   0   0  -1   0 0        4
## [3,]   0  0  0  0  0   0  -1   1   0   1  -1 0        4
## [4,]   0  0  0  0  0  -1   1   0   1  -1   0 0        6
## [5,]  -1  0  0  1  0   0   1   0   0   0   0 0       20
## [6,]  -1  0  0  0  1   0   1   0   0  -1   1 0       18
## [7,] 100  0  0  0  0 500 200   0   0 200 400 1    20800
```

This is the solution to the transportation problem, all in 15 lines of code. We find that the decision for the flows should be:

Factory	Store A	Store B	Store C	Production
Factory I	500	200	0	700
Factory II	0	200	400	600
Demand	500	400	400	

There is a surplus of 100 units at Factory I that we do not need to move (we moved 700 units out of 800 available), and the total cost of distribution is 20,800 (cents) compared to 25,800 and 21,200 in our sample decisions at the beginning.

An important aspect of the code above is that it solves the problem *iteratively*, one step at a time, and it doesn't even care about how large the simplex tableau is. In this way, the above code is the solution to *every* transportation problem.