```
julia> include("ssp_example1.jl")
The shortest path is:[1,3,5]
x vector:[0,1,0,0,1,0,0,0]
Cost is 7.0
Cost is 7.0
Cost is [7.0]
```

One may wish to see the x vector with the start and end nodes:

```
julia> [start_node end_node x]
8x3 Array{Int64,2}:
 1  2  0
 1  3  1
 2  3  0
 3  4  0
 3  5  1
 4  1  0
 4  5  0
 5  2  0
```

6.4 Implementing Dijkstra's Algorithm

In the previous section, we have used the `Graphs.jl` package to solve the shortest path problem. In this section, we will implement Dijkstra's algorithm by ourself. Of course, the code will not be as efficient as the `Graphs.jl` package. Our own implementation will be a good practice of Julia programming and it may be a good reference for implementing other label-setting or label-correction algorithms.

The origin node is o and the destination node is d. Let's look at the algorithm description:

- **Step 0.** Initialize: $w_1 = 0$ and $\mathcal{X} = \{o\}$.
- **Step 1.** When $\overline{\mathcal{X}} = \mathcal{N} \setminus \mathcal{X}$, find the set
$$(\mathcal{X}, \overline{\mathcal{X}}) = \{(i,j) : i \in \mathcal{X}, j \in \overline{\mathcal{X}}\}$$
- **Step 2.** Find a link (p,q) such that
$$w_p + c_{pq} = \min\left\{w_i + c_{ij} : (i,j) \in (\mathcal{X}, \overline{\mathcal{X}}) \text{ and } (i,j) \in \mathcal{A}\right\}$$

- **Step 3.** Set $w_q = w_p + c_{pq}$ and add the node q to the set $\mathcal{X}$.
- **Step 4.** If the new $\overline{\mathcal{X}} = \mathcal{N} \setminus \mathcal{X}$ is an empty set, stop. Otherwise, go to Step 1 and repeat.

Alternatively, in Step 4, one can terminate the algorithm once the destination node d is labeled, i.e. when $q = d$, when the shortest path to one destination is of interest. For the details of the algorithm see Bazaraa et al. (2011)[1] or Ahuja et al. (1993)[2].

If we look at the algorithms, we basically need the set of nodes $\mathcal{N}$, the set of links $\mathcal{A}$, and c_{ij} as input data. As described in the previous section, we have `start_node`, `end_node`, and `c` for these data. The algorithm will update the values of a vector $\mathbf{w}$ and the two sets $\mathcal{X}$ and $\overline{\mathcal{X}}$.

Let's first prepare variables. Here is $\mathcal{N}$:

```
julia> N = Set(1:no_node)
Set([4,2,3,5,1])
```

where `Set` is used. We can think of `Set` as a special type of arrays. This literally represents a set and is useful when the order of elements saved in the array is unimportant. Next, we prepare $\mathcal{A}$ and $\mathbf{c}$ as a dictionary:

```
A = Set{Tuple{Int,Int}}()
c_dict = Dict()
for i in 1:no_link
        push!(A, (start_node[i], end_node[i]))
        c_dict[(start_node[i], end_node[i])] = c[i]
end
```

The set `A` looks like:

[1]Bazaraa, M.S., Jarvis, J.J. and Sherali, H.D., 2011. Linear programming and network flows. John Wiley & Sons.

[2]Ahuja, R.K., Magnanti, T.L. and Orlin, J.B., 1993. Network flows: Theory, algorithms, and applications. Prentice Hall

```
julia>  A
Set([(3,5),(4,5),(1,2),(2,3),(5,2),(4,1),(1,3),(3,4)])
```

Then, we prepare **w** as follows:

```
w = Array(Float64, no_node)
```

where we did not assign any value yet. It will have some default values.

In Step 0, we initialize as follows:

```
w[origin] = 0
X = Set{Int}({origin})
Xbar = setdiff(N, X)
```

where we conveniently computed $\overline{\mathcal{X}} = \mathcal{N} \setminus \mathcal{X}$ using `setdiff`.

For the iterations in Steps 1 to 4, we will use a `while`-loop as follows:

```
while !isempty(Xbar)
        # Step 1
        # Step 2
        # Step 3
        # Step 4
end
```

The statement `isempty(Xbar)` will return `true` if there is no element in `Xbar`, i.e. $\overline{\mathcal{X}} = \emptyset$. Therefore, the above `while`-loop will repeat the steps while `Xbar` is not an empty set and terminates when it is empty.

In Step 1, we find the set $(\mathcal{X}, \overline{\mathcal{X}}) = \{(i,j) : i \in \mathcal{X}, j \in \overline{\mathcal{X}}\}$, named as `XX`:

```
XX = Set{Tuple{Int,Int}}()
for i in X, j in Xbar
        if (i,j) in A
                push!(XX, (i,j))
        end
end
```

where the two `for`-loops are used. Whenever we find a link from `X` to `Xbar`, we add it to `XX`.

In Step 2, we find

$$w_p + c_{pq} = \min\left\{w_i + c_{ij} : (i,j) \in (\mathcal{X}, \overline{\mathcal{X}}) \text{ and } (i,j) \in \mathcal{A}\right\}$$

as follows:

```
min_value = Inf
q = 0
for (i,j) in XX
        if w[i] + c_dict[(i,j)] < min_value
                min_value = w[i] + c_dict[(i,j)]
                q = j
        end
end
```

where `min_value` is used to record $w_p + c_{pq}$ and `q` to record q. Note that `min_value` is initialized as ∞ and updated during the search of the minimum.

Step 3 is as simple as:

```
w[q] = min_value
push!(X, q)
```

Step 4 simply updates `Xbar`:

```
Xbar = setdiff(N, X)
```

After this update of `Xbar`, the `while`-loop will evaluate the condition `!isempty(Xbar)` again, to determine to continue or stop. When the `while`-loop stops, we obtain `w[destination]` as the length of the shortest path from origin to destination.

The complete code is provided:

Listing 6.6: An implementation of Dijkstra's algorithm
code/chap6/ssp_example2.jl

```
# Data Preparation
network_data_file = "simple_network.csv"
network_data = readcsv(network_data_file,  header=true)
data = network_data[1]
header = network_data[2]

start_node = round(Int64, data[:,1])
end_node = round(Int64, data[:,2])
c = data[:,3]

origin = 1
destination = 5

# number of nodes and number of links
no_node = max( maximum(start_node), maximum(end_node) )
no_link = length(start_node)

# Preparing sets and variables
N = Set(1:no_node)

A = Set{Tuple{Int,Int}}()
c_dict = Dict()
for i in 1:no_link
        push!(A, (start_node[i], end_node[i]))
        c_dict[(start_node[i], end_node[i])] = c[i]
end

w = Array(Float64, no_node)

# Step 0
w[origin] = 0
X = Set{Int}([origin])
Xbar = setdiff(N, X)

# Iterations for Dijkstra's algorithm
while !isempty(Xbar)
        # Step 1
        XX = Set{Tuple{Int,Int}}()
        for i in X, j in Xbar
```

```
                if (i,j) in A
                        push!(XX, (i,j))
                end
        end

        # Step 2
        min_value = Inf
        q = 0
        for (i,j) in XX
                if w[i] + c_dict[(i,j)] < min_value
                        min_value = w[i] + c_dict[(i,j)]
                        q = j
                end
        end

        # Step 3
        w[q] = min_value
        push!(X, q)

        # Step 4
        Xbar = setdiff(N, X)
end

println("The length of node $origin to node $destination is: ", w[destination])
```

The result is:

```
julia> include("ssp_example2.jl")
The length of node 1 to node 5 is: 7.0
```