Using R to Create and Solve Assessment Problems for an Online Math for General Education Course

Author: Grace Kelting
Faculty Advisor: Dr. Kristi Karber
Department of Mathematics and Statistics, University of Central Oklahoma

INTRODUCTION

Perhaps one of the most difficult tasks for teachers is creating appropriate assessment problems to ask their students. Additionally, it may be more difficult to create a large library of questions in order to avoid cheating and provide students multiple attempts on a particular type of problem to deepen their understanding. I have used the mathematical program **R** to alleviate some of the struggles a teacher may face creating these problems. Specifically, I have written programs that create and solve problems involving apportionment and graph theory for the University of Central Oklahoma's *Math for General Education* online course. Within the programs is a problem creator that builds a question using randomized numbers and solves it using a specified algorithm.

ACKNOWLEDGEMENTS

This project is based on work supported by a faculty on-campus grant from the Office of Research & Sponsored Programs, University of Central Oklahoma. Special thanks to Jonathan Yarborough and Dr. Sean Laverty for their help and support in R.

GRAPH THEORY

- The **R** code specifically creates and solves problems involving graphs with **Hamiltonian Circuits.**
  - **Hamiltonian:** path visits every vertex once with no repeats.
  - **Circuit:** path begins and ends at the same vertex.
- The **`build.graph`** function allows the user to choose the max potential edge value and path solving method.

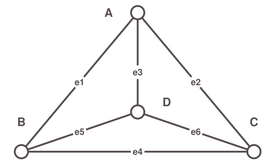
```
build.graph <- function(max.value, method){…}
```

- The solving methods are: Nearest Neighbor Algorithm (**`NNA`**); Repeated Nearest Neighbor (**`RNNA`**); and the Sorted Edges Algorithm (**`SEA`**).
- Using a random 3-digit seed number and the sample command, edge weights are assigned randomly so that every problem created is unique.

```
set.seed(sample(100:999, 1))
n.edges <- 6
edges <- sample(1:max.value, n.edges)
```

- A "connections" matrix keeps track of the neighboring vertices, their connected edges, and the edge weights.

```
$connections
      A     B     C     D
A 10000    e1    e2    e3
B    e1 10000    e4    e5
C    e2    e3 10000    e6
D    e4    e5    e6 10000
```



NEAREST NEIGHBOR ALGORITHM

- Given a starting vertex and the connections matrix, the **`nearest.neighbor`** function finds the "best" Hamiltonian Circuit by going to the neighboring vertices whose edge weight is the smallest (i.e. the "nearest neighbor"). It will only visit that neighbor if that vertex has not yet been visited.

```
nearest.neighbor <- function(vertex, connections){…}
```

- A stack of vertices is created that will eventually become the "best" path. The **`stack`** is initialized as the **`vertex`** passed from the **`build.graph`** function. A **`total.weight`** value of the path is also initialized.

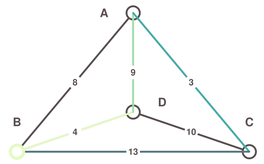
```
stack <- c(vertex)
total.weight <- 0
```

- The neighborhood of the last vertex in the **`stack`** is checked for its "best" neighbor (the one with the smallest edge weight). If the "best" neighbor is already in the **`stack`** (has been visited), then the next "best" neighbor is used.
- The stack is updated with the new vertex and the **`total.weight`** value is updated with the new edge weight.
- The code will run until all of the vertices have been visited and the path returns to the starting vertex.

```
$starting.vertex
[1] "B"

$path
[1] "B" "D" "A" "C" "B"

$total.weight
[1] 29
```



SORTED EDGES ALGORITHM

- Given the list of vertices and the connections matrix, the **`sorted.edges`** function finds the "best" Hamiltonian Circuit by selecting the smallest unused edge in the graph. It will not use an edge if it creates a circuit that does not contain all vertices or gives a vertex of degree 3.

```
sorted.edges <- function(vertices, connections){…}
```

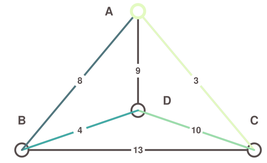
- Two vectors, **`OutputFirst`** and **`OutputLast`**, are initialized to hold the vertices of the "best" edges. The edge weights are compared against each other before the "best" one is chosen. One vertex of the "best" edge is placed in **`OutputFirst`** and the other vertex in **`OutputLast`**.
- The code will run until all of the vertices have been visited and the circuit is complete.

```
> OutputFirst
[1] "A" "B" "A" "C"
> OutputLast
[1] "C" "D" "B" "D"
```

- The first edge's vertices are placed on a stack. The path is then organized based off the last vertex in the stack using the matching vertices in the vectors, **`OutputFirst`** and **`OutputLast`**. The total weight of the path is also calculated.

```
$path
[1] "A" "C" "D" "B" "A"

$total.weight
[1] 25
```



APPORTIONMENT

- Apportionment determines how to divide a fixed number of things among groups of different sizes (e.g. representative seats and populations).
- The **`build.apportionment`** function allows the user to choose the number of populations needing apportioned and the apportionment solving method.

```
build.apportionment <- function(n.places, method){...}
```

- Solving methods: Hamilton, Jefferson, and Huntington-Hill.
- Using a random 3-digit seed number and the sample command, the number of seats and population sizes are assigned randomly so that every problem created is unique.

```
set.seed(sample(100:999, 1))
n.seats <- sample(50:100, 1)
populations <- sample(seq(from = 25000, to = 100000,
    by = 500), n.places)
```

- The original divisor is computed by dividing the total population by the number of seats.

```
divisor <- pop.top["Total","Population"] / n.seats
```

- A population matrix tracks information for all methods.

```
      Population Quota Lower.Quota Geometric.Mean Initial Final (Extra for Hamilton)
A             P1    NA          NA             NA      NA    NA                    NA
B             P2    NA          NA             NA      NA    NA                    NA
C             P3    NA          NA             NA      NA    NA                    NA
D             P4    NA          NA             NA      NA    NA                    NA
E             P5    NA          NA             NA      NA    NA                    NA
Total   PopTotal    NA          NA             NA      NA    NA                    NA
```

HAMILTON'S METHOD

- The **`hamilton`** function uses the population table, divisor, number of seats, and number of populations to determine the allocation of the seats.

```
hamilton <- function(pop.top, div, n.seats, n.places){…}
```

- The floor of the quotas is computed for each population and placed in the **`initial`** column of the population table. The lower quotas are then summed.

```
low.quota <- floor(quota)
pop.top[i,"Initial"] <- low.quota
pop.top["Total","Initial"]
    <- sum(pop.top[c(1:n.places),"Initial"])
```

- If the initial total is not equal to the number of seats needing apportioned, then the extra seats are allotted to the populations based off their quota's decimal values.
- The **`(Extra for Hamilton)`** column contains the truncated decimals for each population quota. The maximum decimal value is determined and its population receives +1 seat. The max value is then changed to -1 to signify the decimals have been used. This code will run until the number of seats is met.

```
[1] "Number of Seats"
[1] 87
[1] "Divisor"
[1] 3258.621
      Population     Quota Initial Final (Extra for Hamilton)
A          31000  9.513228       9    10           -1.00000000
B          78500 24.089947      24    24            0.08994709
C          40000 12.275132      12    12            0.27513228
D          77000 23.629630      23    24           -1.00000000
E          57000 17.492063      17    17            0.49206349
Total     283500        NA      85    87                   NA
```

JEFFERSON'S METHOD

- The **`jefferson`** function uses the population table, divisor, number of seats, number of populations, and a modified divisor list to determine the allocation of seats.

```
jefferson <- function(pop.top, div, n.seats, n.places,
    mod.div.list){…}
```

- The floor of the quotas is computed for each population and placed in the **`initial`** column of the population table and then summed.
- If the initial total is less than the number of seats needed, then the divisor is reduced and the quotas and allocations are recalculated. If the initial total is greater than the number of seats needed, then the divisor is increased. The new divisors are kept track of in the **`mod.div.list`** until a working divisor is found.
- The **`range.jefferson`** and **`test.jefferson`** functions determine the range of working divisors for Jefferson's method. The divisors being tested are tracked in a list with the first working divisor as one bound of the range and the last working divisor as the other bound. Each bound has an error of being <1 away from the actual divisor boundary because the divisors are checked in increments of 1.

```
[1] "Number of Seats"
[1] 87
[1] "Divisor"
[1] 3258.621
[1] "Original Apportionment Table"
      Population     Quota Initial
A          31000  9.513228       9
B          78500 24.089947      24
C          40000 12.275132      12
D          77000 23.629630      23
E          57000 17.492063      17
Total     283500        NA      85
```

```
$`New Apportionment Table`
      Population     Quota Initial
A          31000  9.803317       9
B          78500 24.824528      24
C          40000 12.649441      12
D          77000 24.350174      24
E          57000 18.025453      18
Total     283500        NA      87
$`New Divisor Used`
[1] 3162.195
[1] "Range of Divisors"
[1] 3140.621 3166.621
```

HUNTINGTON-HILL METHOD

- The **`huntington.hill`** function uses the population table, divisor, number of seats, number of populations, and a modified divisor list to determine the allocation of seats.

```
huntington.hill <- function(pop.top, div, n.seats,
    n.places, mod.div.list){…}
```

- The floor of the quotas, n, is computed for each population and placed in the **`Lower.Quota`** column of the population table. The geometric mean, $\sqrt{n(n+1)}$, is calculated and placed in the **`Geometric.Mean`** column of the population table.
- If the quota is larger than the geometric mean, then the quota is rounded up. If it is smaller, then it is rounded down. The resulting quotas are the initial allocation of the seats.
- If the initial total is less than the number of seats needed, then the divisor is reduced. If the initial total is greater than the number of seats needed, then the divisor is increased. Quotas and allocations are then recalculated.
- The **`range.huntington.hill`** and **`test.huntington.hill`** functions determine the range of working divisors for Huntington-Hill. If the original divisor works, **`"it's fine"`**.

```
[1] "Number of Seats"
[1] 87
[1] "Divisor"
[1] 3258.621
[1] "Original Apportionment Table"
      Population     Quota Lower.Quota Geometric.Mean Initial
A          31000  9.513228          9      9.486833      10
B          78500 24.089947         24     24.494897      24
C          40000 12.275132         12     12.489996      12
D          77000 23.629630         23     23.494680      24
E          57000 17.492063         17     17.492856      17
Total     283500        NA         NA             NA      87
$`New Divisor Used`
[1] "it's fine"
```