Project 1: Numerical Wheel

Design Due Date: Friday, February 21st @ 11:59pm
Implementation Due Date (on-time): Wednesday, March 5th @ 11:59pm
Implementation Due Date (late): Thursday, March 6th @ 11:59pm

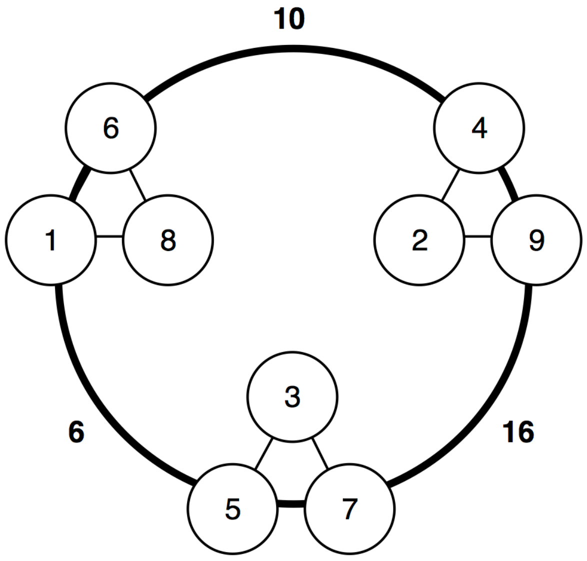

Figure 1: A number wheel composed of three triads

The Problem

A numerical wheel is composed of one or more triads of connected circles whose contents are usually assumed to be initially empty. The goal of the puzzle is to correctly fill the empty circles with numbers. In order to satisfy the problem, the solution must adhere to the following restrictions:

- Each circle is to be filled with a unique integer (from 1 to the number of circles). In the example above, the integers 1 through 9 are used.
- The sum of the three numbered circles in each triad must have the same value. In the example above, each triad sums independently to the same value, 15.
- Each triad is connected on both sides by a bridge that has another value. The adjacent circles in the two triads connected by the bridge must sum together to equal the bridge value. In the example above, 6+4=10 (top), 9+7=16 (bottom right) and 5+1=6 (bottom left).

Input Specification

The input for the numerical wheel comes from standard input and contains two lines, with an optional third line.

- The first line is an integer that indicates the number of triads in the wheel. It is guaranteed to be greater than or equal to 1.
- The second line contains each of the bridge values, as a series of integers,

separated by spaces. It is guaranteed that the number of bridge values will be the same as the number of triads.

- If a third line is not present, it is assumed that the wheel is empty and no circles have prior values associated with them. If a third line is present, it indicates what values are required to be in certain circles for a valid solution. The numbers are all integers specified in pairs:
    - The first number in the pair indicates a unique circle in the wheel. 0 represents the first circle, 1 represents the second circle, and #_circles - 1) represents the last circle. This integer is guaranteed to be in the valid range.
    - The second number in the pair is the circle's value. It is guaranteed to be a value between 1 and the number of circles, inclusive. Also, it is guaranteed that no number in that range will be repeated for different circles.

Sample input

This is the input that corresponds to the above figure, assuming it is currently filled with all values (a solution!):

```
3
10 16 6
0 1 1 6 2 8 3 2 4 4 5 9 6 7 7 3 8 5
```

This is the input for the same wheel that assumes all of the circles are empty:

```
3
10 16 6
```

Output Specification: Non-path mode

By default, the solution, if one exists, should be displayed to standard output. Each triad is displayed in the format `#.#.#`. The bridge connections between the triads are displayed with dashes. This is the output that would correspond to the above figure:

```
1.6.8 - 2.4.9 - 7.3.5
```

If there is no solution to the problem, the following message should be displayed to standard output:

```
No solution!
```

There may be more than one solution that satisfies the problem. Your program only needs to produce one solution.

Output Specification: Path mode

If the program is run in "path mode", and a solution exists, the "series" of steps that lead you from the initially empty configuration to the solution, with one new number added to a circle at each step. Using the figure above, and assuming the wheel is initially empty, the output would be:

```
1.0.0 - 0.0.0 - 0.0.0
1.6.0 - 0.0.0 - 0.0.0
1.6.8 - 0.0.0 - 0.0.0
```

```
1.6.8 - 2.0.0 - 0.0.0
1.6.8 - 2.4.0 - 0.0.0
1.6.8 - 2.4.9 - 0.0.0
1.6.8 - 2.4.9 - 7.0.0
1.6.8 - 2.4.9 - 7.3.0
1.6.8 - 2.4.9 - 7.3.5
```

Again, your solution may be different and would result in different output. Also, the order you fill in the circles is implementation dependent.

Explanation

This type of puzzle can be solved using the classical algorithm known as backtracking. It is a recursive algorithm that starts with a valid configuration (e.g. empty) and produces either a solution, or an indication that no solution exists.

This is the backtracking algorithm expressed in Python code:

```
def solve(configuration):
    """solve: Config -> Config or None"""

    if isGoal(configuration):
        return configuration
    else:
        for child in getSuccessors(configuration):
            if isValid(child):
                solution = solve(child)
                if solution != None:
                    return solution
        return None
```

The backtracking algorithm requires three routines based on the puzzle being solved:

- `isGoal`: A boolean function that indicates whether a valid configuration is a solution, or not.
- `getSuccessors`: A function that takes a valid configuration and produces a collection of successor configurations. In this design, the successors that are generated are exhaustive with respect to possible next steps and are not always guaranteed to be valid.
- `isValid`: A boolean function that indicates whether a configuration is valid or not.

Design Requirements

For this part of the project, you are required to separate the backtracking algorithm from the wheel puzzle. It is ok for the algorithm to "talk" directly to the wheel puzzle. In the future, the algorithm must work with any type of puzzle/configuration.

You will be required to model your class diagrams using UML. If you are unfamiliar with UML, here are a few tutorial sites:

- http://www.tutorialspoint.com/uml
- http://www.sparxsystems.com/resources/uml2_tutorial/index.html
- http://edn.embarcadero.com/article/31863

There are a variety of tools that you can use to generate your class diagrams. Choose your favorite.

- Visual Studio
- StarUML
- ArgoUML
- Lucid Chart
- UMLet
- Rational Rose

Program Requirements

The program must be written entirely in C++, and compile and run on our CS Ubuntu machines using gcc 4.8.2.

The main program is called `wheel`.

- If run with no other arguments, the program is assumed to be in the default mode (no path). It will read the input, and output only the solution, if one exists.
- If run with one argument, the string `"path"`, the program is assumed to be in path mode. It will read the input, and output the solution path, if one exists.
- If run with one argument that is not the string `"path"`, or run with multiple arguments, it will display the following message to *standard error* and exit: `Usage: wheel {path}`

You must do a reasonable amount of pruning in order to solve this puzzle from the 2001 World Puzzle Championship (in Brno, Czech Republic, and attended by esteemed colleague, Professor Zack Butler, for whom you can thank for this puzzle). Your program must solve this problem in under 1 minute on our CS Ubuntu machines:

```
7
30 32 33 28 35 27 18
```

Your program must manage memory correctly, with no invalid memory accesses or leaks. You can use `valgrind` on our CS Ubuntu machines to verify your program:

```
valgrind --leak-check=full wheel
```

For reference, a solution program can be run from:

```
~pskillsCPP/pub/Proj/01-Wheel/wheel
```

Grading

The grade breakdown for this assignment is:

- Functionality: 70%
- Design: 20%
- Memory Management: 5%
- Code Style & Commenting: 5%

A late implementation submission will be graded with a 20% penalty.

Submissions

There are two separate submissions for this assignment. Take note of the separate due dates at the top.

- Submit your class diagram as an image file (preferably PNG) to the MyCourses dropbox, before the deadline.
- Submit your final implementation and final class diagram, in a zip file, to the MyCourses dropox, before the deadline.

---

updated: Fri Feb 14 2014