Red Black Graph - A DAG of Multiple, Interleaved Binary Trees

Daniel W Rapp

December 24, 2021

1 Introduction

When representing binary relationships, such as familial relationships, a number of approaches may be utilized including: ad-hoc, general graphs, specialized tables or charts. As I've explored different representations, a specialized mathematical representation has emerged. This mathematical representation is useful as the underpinnings of dynamic graph algorithms for use cases that include graph analysis, relationship calculation, loop detection, loop prevention, etc.

The underlying mathematical model is a directed, acyclic graph of multiple, interleaved binary trees, designated as a Red Black Graph. The name, Red Black Graph, derives from superficial similarity to Red Black Trees. Red Black Trees are binary trees such that each node has an extra, color bit (red or black). This color bit is used to balance the tree as modifications are made. In a Red Black Graph each vertex also has a an extra, color bit, rather than utilizing the color bit for balancing, the color bit is used to constrain edges between vertices.

I will provide a formal definition of a Red Black Graph, as well as explore a number of interesting emergent properties. I will also examine several applications of Red Black Graphs to illustrate the utility of using this mathematical model for familial relationships.

1.1 Formal Definition

A Red Black Graph is a network, $\mathscr{N}$, consiting of a directed graph, $G = (V, E)$, and a relationship function, $r(\mathbf{u}, \mathbf{v}) \rightarrow \{\mathbb{N}, -1\}$. Additionally, the following constraints are in place:

1. Any given vertex must have a color, either red or black
2. Any given vertex can have at most one outbound edge to a vertex of a given color
3. Every vertex has an edge to itself where:
$$r(\mathbf{v}, \mathbf{v}) \rightarrow \begin{cases} -1, & \text{if } \mathbf{v} \text{ is a red vertex,} \\ 1, & \text{if } \mathbf{v} \text{ is a black vertex} \end{cases}$$
4. If there is no path in G from $\mathbf{u}$ to $\mathbf{v}$, $r(\mathbf{u}, \mathbf{v}) \rightarrow 0$
5. For all other $\mathbf{u}$, $\mathbf{v}$ not covered by constraints 3 and 4, $r(\mathbf{u}, \mathbf{v}) \rightarrow x$, were x is generated by walking the shortest path (assume uniform edge weights) from $\mathbf{u}$ to $\mathbf{v}$ as follows:
6. $x \leftarrow 1$
7. walk the edge along the shortest path and update x as follows ($\ll$ is bitwise shift left):
$$x \leftarrow \begin{cases} x \ll 0, & \text{if resultant vertex is red,} \\ x \ll 1, & \text{if resultant vertex is black} \end{cases}$$

8. repeat B and terminate after **v** is reached

1.2 Motivation

The relationships resulting from sexual reproduction can be modeled by a Red Black Graph, arbitrarily assigning vertices that are male as Red and vertices that are female as Black with direction of edges being from the offspring to the parent.

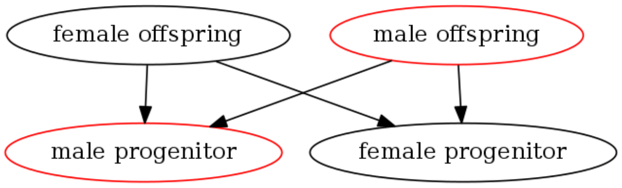

Figure 1: Simple Red Black Graph

1.3 Observation

For a given vertex in a Red Black graph there are two distinct sub-graphs or "views" or perspectives, for a given vertex: Descendency and Ancestry.

Descendency is the sub-graph for a given vertex that consists of all vertices and edges that can follow a graph traversal and arrive at the given node.

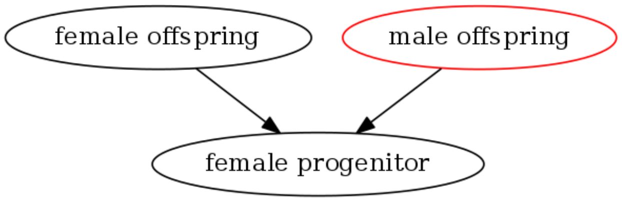

Figure 2: Descendency View for Female Progenitor

Ancestry is the sub-graph for a given vertex that consists of all the vertices and edges reachable by following out-bound edges. This sub-graph is a well-formed binary tree.

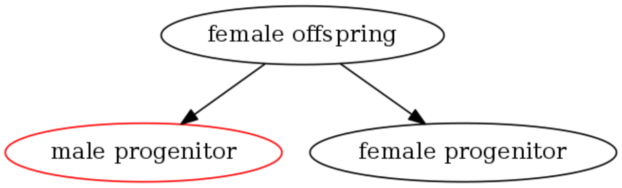

Figure 3: Ancestry View for Female Offspring

The Ancestry sub-graph from a given **u** is a well-formed binary tree. $r(\mathbf{u}, \mathbf{v})$ is defined as to number the nodes of the tree as they are encountered in a breadth first search. An concrete example of $r(\mathbf{u}, \mathbf{v})$ is observed in a pedigree chart (common in family history research). See fig. "Pedigree Chart".

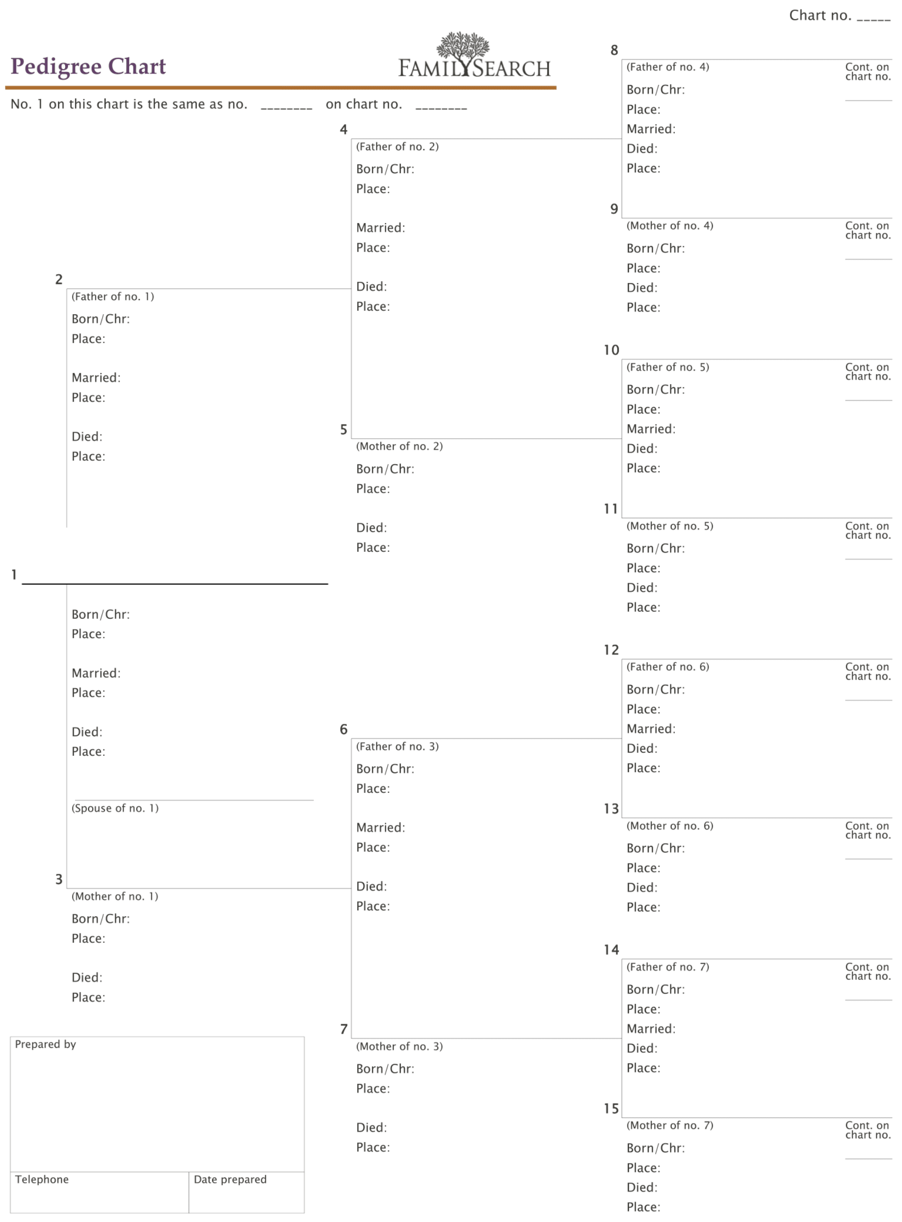

Chart no. _____

Pedigree Chart

No. 1 on this chart is the same as no. ________ on chart no. ________

1 ______________________________
Born/Chr:
Place:
Married:
Place:
Died:
Place:
(Spouse of no. 1)

2
(Father of no. 1)
Born/Chr:
Place:
Married:
Place:
Died:
Place:

3
(Mother of no. 1)
Born/Chr:
Place:
Died:
Place:

4
(Father of no. 2)
Born/Chr:
Place:
Married:
Place:
Died:
Place:

5
(Mother of no. 2)
Born/Chr:
Place:
Died:
Place:

6
(Father of no. 3)
Born/Chr:
Place:
Married:
Place:
Died:
Place:

7
(Mother of no. 3)
Born/Chr:
Place:
Died:
Place:

8
(Father of no. 4) Cont. on chart no.
Born/Chr:
Place:
Married:
Died:
Place:

9
(Mother of no. 4) Cont. on chart no.
Born/Chr:
Place:
Died:
Place:

10
(Father of no. 5) Cont. on chart no.
Born/Chr:
Place:
Married:
Died:
Place:

11
(Mother of no. 5) Cont. on chart no.
Born/Chr:
Place:
Died:
Place:

12
(Father of no. 6) Cont. on chart no.
Born/Chr:
Place:
Married:
Died:
Place:

13
(Mother of no. 6) Cont. on chart no.
Born/Chr:
Place:
Died:
Place:

14
(Father of no. 7) Cont. on chart no.
Born/Chr:
Place:
Married:
Died:
Place:

15
(Mother of no. 7) Cont. on chart no.
Born/Chr:
Place:
Died:
Place:

Prepared by

Telephone

Date prepared

Figure 4: Pedigree Chart

1.4 Adjacency Matrix

An adjacency matrix is a square matrix used to represent a graph. The elements of the matrix are 1 (or edge weight) if there is an edge between the vertices represented by the column index and the row index. Slightly more formally, for G with a vertex set V, the adjacency matrix is a square $|V|$ x $|V|$ matrix, A, such that A_{ij} is 1 if there exists an edge from **i** to **j** and 0 otherwise.

Given the above example graph and chosing indices for the vertices as follows: 0 - Female Offspring, 1 - Male Offspring, 2 - Male Progenitor, 3 - Female Progenitor, the graph would be represented by the following adjacency matrix.

$$A = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

1.5 Red Black Graph Adjacency Matrix

I'll define the adjacency matrix for a Red Black Graph only slightly differently. $R_{ij} = r(\mathbf{i}, \mathbf{j})$ if there is an edge from **i** to **j** and 0 otherwise. The following is the Red Black adjacency matrix for the above example graph.

$$R = \begin{bmatrix} 1 & 0 & 2 & 3 \\ 0 & -1 & 2 & 3 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Observe the following properties:

$$trace(R) = |V_{black}| - |V_{red}|$$

$$|V| = |V_{black}| + |V_{red}|$$

$$|V_{black}| = \frac{|V| + trace(R)}{2}$$

$$|V_{red}| = \frac{|V| - trace(R)}{2}$$

1.6 Transitive Closure

Computing the transitive closure of an adjacency matrix, A, results in the reachability a matrix, A^+, that shows all vertices that are reachable from any given vertex. If $A_{ij} == 1$ there is a path from $\mathbf{v}_i$ to $\mathbf{v}_j$.

The transitive closure of a Red Black adjacency matrix, R, is defined to be the relationship matrix, R^+. In addition to reachability, R^+ is defined in such that if $R^+_{ij} ==$ **n** and **n** is non-zero, then from **n** we can derive both the path lengh and the explicit traversal path from **i** to **j**.

As an example consider the following graph, where each node has been labeled with a vertex index:

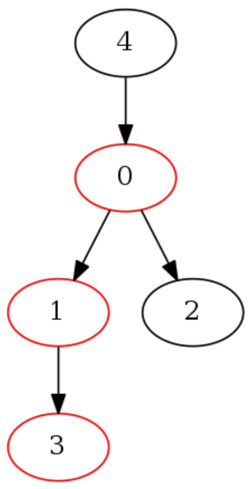

Figure 5: Red Black Graph Example for Transitive Closure

By inspection:

$$R = \begin{bmatrix} -1 & 2 & 3 & 0 & 0 \\ 0 & -1 & 0 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \\ 2 & 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } R^+ = \begin{bmatrix} -1 & 2 & 3 & 4 & 0 \\ 0 & -1 & 0 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \\ 2 & 4 & 5 & 8 & 1 \end{bmatrix}$$

Before examining how to generate R^+ from R, consider the following observations:

- $r(\mathbf{u}, \mathbf{v})$ is even when $\mathbf{v}$ is red and odd when black. (By convention -1 is considered even.)
- The path length is inherent in $r(\mathbf{u}, \mathbf{v})$, and is trivially derived by taking the integral portion of $log_2(r(\mathbf{u}, \mathbf{v}))$.
- The traversal path is also inherent in $r(\mathbf{u}, \mathbf{v})$ and can be derived by reversing r's defintion and successively right shifting out bits (of a $base_2$ integer representation) and using that bit to "walk" the traversal edge to a red vertex or black vertex.
- The diameter of R^+ is given by $log_2(max(r(\mathbf{u}, \mathbf{v})))$.

1.7 Red Black Arithmetic

As constructing R^+ by inspection is cumbersome for non-trivial cases, I'll explain a method to algorithmically derive R^+ from R.

1.7.1 Avos Product

The first step requires defining a transitive relationship function. To illustrate, consider 3 vertices: $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$. Further, assume that there is a path from $\mathbf{u}$ to $\mathbf{v}$ and from $\mathbf{v}$ to $\mathbf{w}$. I'll define the function, $f(r(\mathbf{u}, \mathbf{v}), r(\mathbf{v}, \mathbf{w})) \to r(\mathbf{u}, \mathbf{w})$, as the *avos product* and designate it using $\star$, e.g. $r(\mathbf{u}, \mathbf{w}) = r(\mathbf{u}, \mathbf{v}) \star r(\mathbf{v}, \mathbf{w})$.

If $r(\mathbf{u}, \mathbf{v}) = 4$ ($\mathbf{v}$ is $\mathbf{u}$'s paternal grandfather) and $r(\mathbf{v}, \mathbf{w}) = 7$ ($\mathbf{w}$ is $\mathbf{v}$'s maternal grandmother). Using either the defintion of r or transcribing $\mathbf{v}$'s pedigree into the proper place in $\mathbf{u}$'s pedigree chart we see that $r(\mathbf{u}, \mathbf{w}) = 19$, or $19 = 4 \star 7$. To further explore this consider the following ($r(\mathbf{x}, \mathbf{y})$ represented in base-2 for illustrative purposes):

$\mathbf{v}$'s relationship to $\mathbf{u}$	$\mathbf{w}$'s relationship to $\mathbf{v}$	$r(\mathbf{u},\mathbf{v})$	$r(\mathbf{v},\mathbf{w})$	$r(\mathbf{u},\mathbf{w})$
father	father	10	10	100
father	mother	10	11	101
mother	father	11	10	110
mother	mother	11	11	111
father	paternal grandfather	10	100	1000
maternal grandmother	paternal grandfather	111	100	11100

While perhaps not obvious, upon examination of the binary representation, the avos product simply replaces the most significant bit of $r(\mathbf{v},\mathbf{w})$ with the value of $r(\mathbf{u},\mathbf{v})$.

To complete the definiton of the Avos Product, the following conventions are required: 1. $-1 = -1 \star 1$ 2. $-1 = 1 \star -1$ 3. For all other cases, -1 is treated as 1

1.7.2 Avos Sum

It is natural to pair multiplication with addition for linear algebra operations (matrix multiplication, matrix distance product, etc.) a min-avos product pairing conforms to the definition of Red Black Graphs with a slight modification required for implementation in a typical RAM computer, namely $-1 = 2^{\mathbf{w}} - 1$, where $\mathbf{w}$ is the word size of the numeric type. As, aside from -1, the range of $r(\mathbf{u},\mathbf{v})$ is $\mathbb{N}$, it is natural to utilize unsigned numeric types (in a strongly typed language). Additionally, I use the convention $min(0) = \infty$. The min function needs to be aware of both these conventions. These conventions define *avos sum* which is designated by †, e.g. $-1 = -1 \dagger 5$ and $2 = 37 \dagger 2$ and $3 = 0 \dagger 3$.

1.7.3 Reference Implementation of Avos Arithmetic

[ ]:
```
# %load ../redblackgraph/reference/avos.py
from redblackgraph.reference.util import MSB

def avos_sum(x: int, y: int) -> int:
    '''
    The avos sum is the non-zero minumum of x and y
    :param x: operand 1
    :param y: operand 2
    :return: avos sum
    '''
    if x == 0:
        return y
    if y == 0:
        return x
    if x < y:
        return x
    return y

def avos_product(x: int, y: int) -> int:
    '''
    The avos product replaces the left most significant bit of operand 2 with operand 1
    :param x: operand 1
    :param y: operand 2
    :return: avos product
    '''

    # negative values are invalid (aside from -1)
    if x < -1 or y < -1:
        raise ValueError(f"Invalid input. Negative values (aside from -1) are not allowed. x: {x}, y:{y}")
    # The zero property of the avos product
    if x == 0 or y == 0:
        return 0
    # Special case -1 * 1 or -1 * -1
    if x == -1:
        if y == 1:
            return -1
        x = 1
    if y == -1:
        if x == 1:
            return -1
        y = 1

    bit_position = MSB(y)
    return ((y & (2 ** bit_position - 1)) | (x << bit_position))
```

1.8 Transitive Closure for Red-Black Adjacency Matrix

Transitive closure of an adjacency matrix can be computed a number of ways, a simple approach is the Floyd-Warshall Algorithm.

Summarized, this algorithm is a tripple loop across the matrix indices continously updating the current transitive relationship, $A^{+}_{i,j}$, if there is a relationship from $A_{i,k}$ and a relationship from $A_{k,j}$. For R^{+}, the Floyd-Warshall algorithm is modified so that the transitive relationship for $R^{+}_{i,j}$ is defined as $R^{+}_{i,j} = R_{i,j} \dagger R_{i,k} \star R_{k,j}$.

A number of matrix operations utilize a "sum of products" pattern (matrix mutliplication, some versions of Floyd-Warshall, etc.). A similar pattern of "avos sum of avos products" is present in operations on matrix representations of Red Black Graphs.

[ ]:
```
# %load ../redblackgraph/reference/transitive_closure.py
import numpy as np
from typing import Sequence
from redblackgraph.reference.avos import avos_sum, avos_product
from redblackgraph.reference.util import MSB
from redblackgraph.types.transitive_closure import TransitiveClosure


def transitive_closure(M: Sequence[Sequence[int]], copy:bool=True) -> TransitiveClosure:
    '''Computes the transitive closure of a Red Black adjacency matrix and as a side-effect,
    the diameter.'''

    # Modification of stardard warshall algorithm:
    # * Replaces innermost loop's: `W[i][j] = W[i][j] or (W[i][k] and W[k][j])`
    # * Adds diameter calculation
    n = len(M)
    W = np.array(M, copy=copy)
    diameter = 0
    for k in range(n):
        for i in range(n):
            for j in range(n):
                W[i][j] = avos_sum(W[i][j], avos_product(W[i][k], W[k][j]))
                diameter = max(diameter, W[i][j])
    return TransitiveClosure(W, MSB(diameter))
```

The example above where R^{+} was derived by inspection, can now be computed:

[ ]:
```
from redblackgraph.reference import transitive_closure
R = [[-1,  2,  3,  0,  0],
     [ 0, -1,  0,  2,  0],
     [ 0,  0,  1,  0,  0],
     [ 0,  0,  0, -1,  0],
     [ 2,  0,  0,  0,  1]]
transitive_closure(R).W
```

1.9 Observations

Given R^+, observe that:

- row vectors represent the complete ancestry view for a given vertex
- column vectors represent the complete descendency view for a given vertex
- row vectors representing siblings will be identical
- column vectors representing siblings will be independant if either of the siblings have offspring represented in the graph
- determining whether **v** is an ancestor of **u** is $\mathcal{O}(1)$ and provided by $R^+[u, v]$

2 Applications of R^+

2.1 Calculating Relationship Between Two Rows in R^+

With R^+ there exists an efficient way to determining full kinship (see: consanguinity) between any two vertices.

1. Given two row vectors from R^+, $\vec{u}$ and $\vec{v}$, find the minimum of $\vec{u}_i + \vec{v}_i$ where both $\vec{u}_i$ and $\vec{v}_i$ are non-zero. This yields values, $r(\mathbf{u}, \mathbf{i})$ and $r(\mathbf{v}, \mathbf{i})$ expressing the relationship of **u** and **v** to the nearest common ancestor, $\mathbf{i_{min}}$
2. Determine the path length from **u** and **v** to the common ancestor, $log_2(r(\mathbf{u}, \mathbf{i}))$ and $log_2(r(\mathbf{v}, \mathbf{i}))$.
3. Using a Table of Consanguinity, calculate the relationship

2.1.1 Observation

Determining whether **u** is related to **v** is $\mathcal{O}(m)$ where m is the expected number of ancestors and $m << |V|$ (assuming an efficient sparse matrix representation). Empirically, m is on the order of $log_2(|V|)$.

A simple implementation follows:

[ ]:
```python
# %load ../redblackgraph/reference/calc_relationship.py
from typing import Sequence
from redblackgraph.types.relationship import Relationship
from redblackgraph.reference.util import MSB


def lookup_relationship(da: int, db: int) -> str:
    '''
    This is a very rudimentary implementation of a Consanguinity lookup and doesn't handle many
    cases correctly.
    :param da: generational distance from u to common ancestor
    :param db: generational distance from v to common ancester
    :return: a string designating relationship
    '''
    removal = abs(da - db)
    if da == 0 or db == 0:
        # direct ancestor
        if removal == 1:
            return "parent"
        if removal == 2:
            return "grandparent"
        if removal == 3:
            return "great grandparent"
        return f"{removal - 2} great grandparent"
    else:
        # cousin
        generational = min(da, db)
        return f"{generational - 1} cousin {removal} removed"


def calculate_relationship(a: Sequence[int], b: Sequence[int]) -> Relationship:
    '''
    Determine if a relationship exists between u, v where u, v are row vectors of the transitive
    closure of a Red Black adjacency matrix
    :param a: row vector for vertex u
    :param b: row vector for vertex v
    :return: (Relationship designation, common ancestor vertex)
    '''

    common_ancestor, (x, y) = min([e for e in enumerate(zip(a, b))
                                   if not e[1][0] == 0 and not e[1][1] == 0],
                                  key=lambda x: x[1][0] + x[1][1],
                                  default=(-1, (0, 0)))

    if common_ancestor == -1:
        return Relationship(-1, "No Relationship")
    return Relationship(common_ancestor,
                        lookup_relationship(MSB(x), MSB(y)))
```

3 Linear Algebra

3.1 Introduction

Having provided a formal definition for a Red Black Graph, looked at its adjacency matrix, R, the transitive closure of its adjacency matrix, R^+, and the avos sum and product, let's extend these observations into a more general discussion of how principles of linear algebra can be applied to Red Black Graphs.

3.2 Vector Classes

Within the context of a Red Black Graph and its matrix representations, R and R^+, the following vector classes are defined:

- *row* vector - represented as $\vec{\mathbf{u}}$. These vectors represent ancestry for a given vertex. Values for elements in these vectors are constrained to whole numbers and -1 where any number, aside from 0, may appear in an alement at most once and where either -1 or 1 must appear as an element but not both.
- *column* vector - represented by $\vec{\mathbf{v}}$. These vectors represent descendency for a given vertex. Values for elements in these vectors are constrained to whole numbers and -1 where either -1 or 1 must appear as an element but not both. Futhermore if -1 appears as an element any further non-zero integer elements must be even and if 1 appears as an element any further non-zero integer elements must be odd.
- *simple row vector* - represented by $\vec{\mathbf{u}}_s$. Row vectors for which elements are constrained to {-1, 0, 1, 2, 3}. These represent a given vertex and it's immediate ancestry only.
- *simple column vector* - represented by $\vec{\mathbf{v}}_s$. Column vectors for which elements are constrained to {-1, 0, 1, 2, 3}. These represent a given vertex and it's immediate descendency only.
- *closed row vector* - represented by $\vec{\mathbf{u}}_c$. Row vectors from R^+. These represent the complete ancestry for a given vertex.
- *closed column vector* - represented by $\vec{\mathbf{v}}_c$. Column vectors from R^+. These represent the complete descendency for a given vertex.

- *compositional vectors* - represented by $\vec{\mathbf{u}}_s^c$ or $\vec{\mathbf{v}}_s^c$. Compositional vectors conform to the constrainst of simple row or column vectors with the following additional constraint: neiter 1 nor -1 appear as an element. The color of the vector is inherent to the vector but not carried as an element. Any consraints due the color are present as if the color were present as an element. Color, if significant, is represented notationally by replacing the supersscript c with the color designation, either r or b.

3.3 Avos Product for Vectors

Consider what an avos vector product might represent. Given a row vector and a column vector, the avos product is $r(\mathbf{u}, \mathbf{v})$, the relationship between the vertices representing the row and column vectors respectively.

Consider the R from the transitive closure example:

$$\begin{bmatrix} -1 & 2 & 3 & 0 & 0 \\ 0 & -1 & 0 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \\ 2 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The 4th row vector of R is $\vec{\mathbf{u}}_s$ for $vertex_4$ while the 2nd column vector is $\vec{\mathbf{v}}_s$ for $vertex_2$. It is observable by inspection that relationship of $vertex_4$ and $vertex_2$ is $r(\mathbf{u}, \mathbf{v}) == 5$ or:

$$\begin{bmatrix} 2 & 0 & 0 & 0 & 1 \end{bmatrix} \star \begin{bmatrix} 3 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = 5$$

The vector dot product, summing element-wise products, results in a scaler value of 6. Avos summing of element-wise avos products does yield 5, which represents a relationship.

A simple implementation of the avos vector product follows:

[ ]:
```
# %load ../redblackgraph/reference/vec_avos.py
from functools import reduce
from redblackgraph.reference import avos_product, avos_sum

def vec_avos(u, v):
    '''Given two vectors, compute the avos product.'''
    return reduce( avos_sum, [avos_product(a, b) for a, b in zip(u, v)])
```

3.4 Observation - Simple/Complete Relationship

The product of a simple row vector and the transitive closure of a Red Black adjacency matrix is a closed row vector

$$\vec{\mathbf{u}}_s \star R^+ = \vec{\mathbf{u}}_c$$

The product of the transitive closure of a Red Black adjacency matrix and a simple column vector is a closed column vector

$$R^+ \star \vec{\mathbf{v}}_s = \vec{\mathbf{v}}_c$$

TODO: Need to walk through an explanation of why this is so.

3.5 Avos product for Matrices

With scaler and vector avos products defined, extension to matrices is elementary. Given A and B, both matrices following the constraints defined for R, and $C = A \star B$, the elements of C_{ij} are given by the vector avos product of $\mathbf{u}_i$ from A and $\mathbf{v}_j$ from B

Avos matrix multiplication of general matrices seems a little abstract so consider the following practical example. $R \star R$ shows all vertices directly related by following up to 2 relationship edges, $R \star R \star R$ shows all vertices related by following up to 3 relationship edges, etc. For some $m <= |V|$ there will be a $\prod_{n=1}^{m} R == R^+$.

A simple implementation of the avos matrix product follows:

```
[ ]: # %load ../redblackgraph/reference/mat_avos.py
     from functools import reduce
     from redblackgraph.reference import avos_product, avos_sum


     def mat_avos(A, B):
         '''Given two matrices, compute the "avos" product.'''
         return [[reduce( avos_sum, [avos_product(a, b) for a, b in zip(A_row, B_col)]) for B_col in zip(*B)]␣
      ↪for A_row in A]
```

3.6 Relational Composition

3.6.1 Adding a Vertex to R^+

Consider the case of adding a new vertex to a red black graph. The new vertex, λ, may introduce edges to/from vertices in the graph and the corresponding row/column vectors conform to the compositional vector classes defined above. Specifically if adding a red vertex to the graph, the vectors $\mathbf{u}^r_{\lambda,s}$ and $\mathbf{v}^r_{\lambda,s}$ define the composition, or if adding a black vertex to the graph, the vectors $\mathbf{u}^b_{\lambda,s}$ and $\mathbf{v}^b_{\lambda,s}$ define the composition. These compositional vectors have non-zero elements only for immediate ancestry/descendency. The operation of adding a new vertex to a graph is designated the "vertex relational composition" and is defined where R^+ is a square matrix of dimension N and R^+_λ is a square matrix of dimension $N+1$ and the colors of $\mathbf{u}^{\vec{c}}_{\lambda,s}$ and $\mathbf{v}^{\vec{c}}_{\lambda,s}$ must be the same. The notation of the vertex relational composition is:

$$R^+_\lambda = \mathbf{u}^{\vec{c}}_{\lambda,s} R^+ \mathbf{v}^{\vec{c}}_{\lambda,s_{color}}$$

The simple/complete relationship observation above can be applied in this instance. $\mathbf{u}^{\vec{c}}_{\lambda,c} = \mathbf{u}^{\vec{c}}_{\lambda,s} \star R^+$ and $\mathbf{v}^{\vec{c}}_{\lambda,c} = R^+ \star \mathbf{v}^{\vec{c}}_{\lambda,s}$.

$\mathbf{u}^{\vec{c}}_{\lambda,c}$ and $\mathbf{v}^{\vec{c}}_{\lambda,c}$ are the row and column, respectively, that need to be appended to R^+ (along with the final diagonal element corresponding to λ's color) to compose R^+_λ. Appending the complete compositional vectors to R^+ isn't sufficient to compose R^+_λ. The "body" of R^+ needs to be "updated" to ensure that R^+_λ is also transitively closed. For each row in R^+, every element in that row is set to the avos product of the corresponding column element in $\mathbf{v}^{\vec{c}}_{\lambda,c}$ and the corresponding row element in $\mathbf{u}^{\vec{c}}_{\lambda,c}$.

Expressing this algorithmically:

1. generate $\mathbf{u}^{\vec{c}}_{\lambda,c} = \mathbf{u}^{\vec{c}}_{\lambda,s} \star R^+$
2. generate $\mathbf{v}^{\vec{c}}_{\lambda,c} = R^+ \star \mathbf{v}^{\vec{c}}_{\lambda,s}$
3. Compose R^+_λ by:

1. appending $\mathbf{u}^c_{\lambda,c}$ to R^+ as a new row
2. appending $\mathbf{v}^c_{\lambda,c}$ to R^+ as a new column
3. setting the diagnoal element $R^+_{\lambda\ N+1,N+1}$ to either 1 or -1 depending on the color of the composition.
4. For each row, i, and each column, j, where $\mathbf{u}^{\vec{c}}_{\lambda,c_j} \neq 0$, set $R^+_{\lambda\ i,j} = \mathbf{u}^{\vec{c}}_{\lambda,c_j} \star \mathbf{v}^{\vec{c}}_{\lambda,c_i}$

3.6.2 Adding an Edge to R^+

[**TODO**: This notation is muddy. Need to rework this section]

Consider the case of adding a new edge to a red black graph. The operation of adding a new edge to a graph is designated the "edge relational composition". The new edge is added between two existing vertices, $vertex_\alpha$ and $vertex_\beta$. The notation of the edge relational composition is:

$$R^+_\lambda = R^+ \star r(\mathbf{ff}, \mathbf{fi})$$

As in the vertex relational composition, we'll make use of the simple/complete relational observation. In this case, the row representing $vertex_\alpha$ is replaced with the avos product of itself (with $element_\beta$ replaced with $r(\mathbf{ff}, \mathbf{fi})$) and R^+. Notationally: $R^{+'} = R^+ +_\alpha ((vertex_\alpha +_\beta r(\mathbf{ff}, \mathbf{fi})) \star R^+)$ where $+_i$ designates replacement of element i in the LHS with the value of the RHS. As in the vertex relational composition, replacing row vector α with it's complete form isn't sufficient to compose R^+_λ. The remainder of the row vectors need to be closed with the new relationship. For each row, i, in $R^{+'}$ excluding α, every element, j in that row is set to $R^{+'}_{i,\alpha} \star R^{+'}_{\alpha,j}$.

Expressing this algorithmically:

1. generate $\vec{\mathbf{u}}'_\alpha = \vec{\mathbf{u}}_\alpha +_\beta r(\mathbf{ff}, \mathbf{fi})$, where $\vec{\mathbf{u}}_\alpha$ is row α in R^+
2. generate $\vec{\mathbf{u}}^{c'}_\alpha = \vec{\mathbf{u}}'_\alpha \star R^+$
3. Compose R^+_λ by:
    1. replacing row α in R^+: $R^{+'} = R^+ +_\beta \vec{\mathbf{u}}^{c'}_a \alpha$
    2. For each row, i, and each column, j, where $i \neq \alpha$, set $R^+_{\lambda\ i,j} = R^{+'}_{i,\alpha} \star R^{+'}_{\alpha,j}$

3.6.3 Simple Implementations

[ ]:
```
# %load ../redblackgraph/reference/rel_composition.py
from redblackgraph.reference.avos import avos_sum, avos_product
from redblackgraph.reference.mat_avos import mat_avos
import copy


def vertex_relational_composition(u, R, v, color):
    '''
    Given simple row vector u, transitively closed matrix R, and simple column vector v where
    u and v represent a vertex, lambda, not currently represented in R, compose R_{lambda}
    which is the transitive closure for the graph with lambda included
    :param u: simple row vector for new vertex, lambda
    :param R: transitive closure for Red Black graph
    :param v: simple column vector for new vertex, lambda
    :param color: color of the node either -1 or 1
    :return: transitive closure of the graph, R, with new node, lambda
    '''
    N = len(R)
    uc_lambda = mat_avos(u, R)
    vc_lambda = mat_avos(R, v)
    R_lambda = copy.deepcopy(R)
    R_lambda.append(uc_lambda[0])
    for i in range(N):
        R_lambda[i].append(vc_lambda[i][0])
        for j in range(N):
            if uc_lambda[0][j] != 0:
                R_lambda[i][j] = avos_sum(avos_product(vc_lambda[i][0], uc_lambda[0][j]), R_lambda[i][j])
    R_lambda[N].append(color)
    return R_lambda

def edge_relational_composition(R, alpha, beta, relationship):
    '''
    Given a transitively closed graph, two vertices in that graph, alpha and beta, and the
    relationship from alpha to beta, compose R'', which is the transitive closure with the
    new edge included
    :param R:
    :param alpha: a vertex in the graph (row index)
    :param beta: a vertex in the grpah (column index)
    :param relationship: r(alpha, beta)
    :return: transitive closure of the grpah, R, with new edge
    '''
    N = len(R)
    u_lambda = [R[alpha]]
    u_lambda[0][beta] = relationship
    u_lambda = mat_avos(u_lambda, R)
    R_lambda = copy.deepcopy(R)
    R_lambda[alpha] = u_lambda[0]
    for i in range(N):
        for j in range(N):
            if R_lambda[alpha][j] != 0:
                R_lambda[i][j] = avos_sum(avos_product(R_lambda[i][alpha], R_lambda[alpha][j]),␣
↪R_lambda[i][j])
    return R_lambda
```

4 Applications of avos Linear Algebra

4.1 Loop Prevention

An issue that can be encountered in systems that represent familial relationships is the inadvertent injection of graph cycles, resulting in the "I am my own Grandpa" case. While this is impossible when relationships model sexual reproduction, the introduction of step-relationships, etc. would make this a possibility. Often times there is ambiguity in the available historical records. If a researcher isn't careful, cylces may result as a genealogical model is created. Modifications to both forms of the relational composition algorithms can prevent the introduction of cycles into the graph.

4.1.1 Vertex Relational Composition Loop Prevention

As vertices are added to an existing graph via relational composition, the intermedite, complete compositional vectors, $\mathbf{u}^{\vec{c}}_{\lambda,c}$ and $\mathbf{v}^{\vec{c}}_{\lambda,s}$ represent the complete ancestry and complete descedency for the new vertex λ respectively. The cycle constraint would be invalidated should there be any vertex that simultaneously appears in the ancestry and descendency for a given vertex.

Given $\mathbf{u}^{\vec{c}}_{\lambda,c}$ and $\mathbf{v}^{\vec{c}}_{\lambda,s}$ of dimension n, the **vertex relational composition** is undefined if there exists a dimension i where $i \neq n \wedge \mathbf{u}^{\vec{c}}_{\lambda,c_i} \neq 0 \wedge \mathbf{v}^{\vec{c}}_{\lambda,s_i} \neq 0$ and is well-formed otherwise.

4.1.2 Edge Relational Composition Loop Prevention

This case is trivial with a transitively closed matrix. Given R^+ and $r(\mathbf{ff}, \mathbf{fi})$, the **edge relational composition** is undefined if $r(\mathbf{fi}, \mathbf{ff}) \neq 0$ and well-formed otherwise.

4.2 Connected Component Identification

[**TODO**: rework this section to tie together topological sort + component identification in a single DFS pass. This will be more efficient than current algorithm.]

As Red Black Graphs are used to represent family relationships, an interesting case is determining how many disjoint trees are represetned within a graph. Tarjan's algorithm is typically used to compute the connected components of a graph. In the case of a transitively closed adjacency matrix, the depth first search used in Tarjan's algorithm is inherently "pre-computed". Because of this property, Tarjan's algorithm can be simplified.

[ ]:
```python
# %load ../redblackgraph/reference/components.py
def find_components(A):
    """
    Given an input adjacency matrix compute the connected components
    :param A: input adjacency matrix (transitively closed)
    :return: a vector with matching length of A with the elements holding the connected component id of
    the identified connected components
    """
    n = len(A)
    u = [0] * n
    component_number = 1
    u[0] = component_number
    for i in range(n):
        if u[i] == 0:
            component_number += 1
            u[i] = component_number
        row_component_number = u[i]
        for j in range(n):
            if A[i][j] != 0:
                if u[j] == 0:
                    u[j] = row_component_number
                elif u[j] != row_component_number:
                    # There are a couple cases here. We implicitely assume a new row
                    # is a new component, so we need to back that out (iterate from 0
                    # to j), but we could also encounter a row that "merges" two
                    # components (need to sweep the entire u vector)
                    for k in range(n):
                        if u[k] == row_component_number:
                            u[k] = u[j]
                    component_number -= 1
                    row_component_number = u[j]
                    u[i] = row_component_number
    return u
```

Consider the following graph

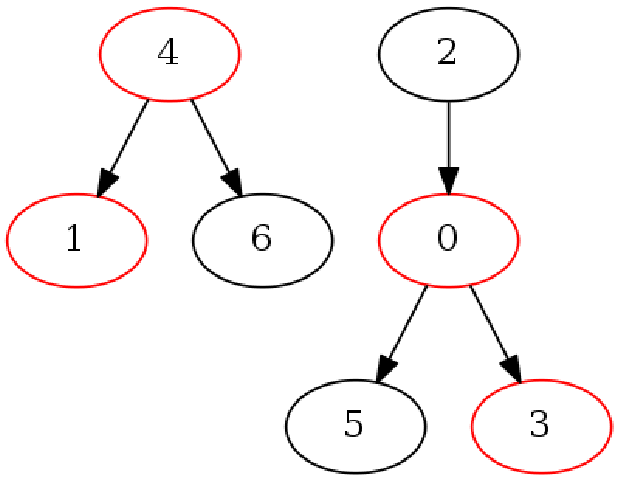

Figure 6: Graph with Components

By inspection, there are two components and the application of the simplified Tarjan's algorithm identifies which vertices belong to which components.

[ ]:
```
from redblackgraph.reference.components import find_components
R = [[-1, 0, 0, 2, 0, 3, 0],
     [ 0,-1, 0, 0, 0, 0, 0],
     [ 2, 0, 1, 0, 0, 0, 0],
     [ 0, 0, 0,-1, 0, 0, 0],
     [ 0, 2, 0, 0,-1, 0, 3],
     [ 0, 0, 0, 0, 0, 1, 0],
     [ 0, 0, 0, 0, 0, 0, 1]]
find_components(R)
```

With an efficient sparse representation this algorithm is also $\mathbf{O}(|V| + |E|)$.

4.3 Canonical Form

Returning to the example, it is obvious from inspection that one component consists of 4 nodes, the other of 3 and that the diamter of the larger component is 2, while the diamter of the smaller is 1. As this information is readily available in the Red Black Graph, it is easily added to the $find\_components$ algorithm (see the following $find\_components\_extended$ algorithm). Observe thatsymetrically permuting a matrix corresponds to relabeling the vertices of the associated graph. I will show that with an appropriate relabeling of the graph vertices the Red Black graph adjacency matrix is upper triangular, R^{+_c} or canonical form. I will also show that $R^{+_c} = \mathbf{P}R^{+}\mathbf{P}^{\top}$ where $\mathbf{P}$ is a permutation matrix derived from the count of the vertices in a component. $\mathbf{P}$ will be chosen such that R^{+_c} in addition to being upper triangular, each graph component and its diameter is readily identified.

To arrive at $\mathbf{P}$ the list of nodes is sorted (in reverse order) first on the size of the encompassing connected component, secondly on the identifier of the connected component and finally on the maximum $r(\mathbf{u}, \mathbf{v})$ for the vertex. The vertices are then labeled based on this sorting, e.g. the $zero^{th}$ vertex is the vetex from the largest connected component that has the greatest $r(\mathbf{u}, \mathbf{v})$ (or most distant ancestor) on down to the n^{th} vertex which is the vertex from the smallest connected component with no (or nearest) ancestor. (Ordering is arbitrary for vertices with identical sort keys.)

A simple implementation of triangularizing R based on the properties inherent in the adjacency matrix and the extended $find\_components$ algorithm follows.

[ ]:
```
# %load ../redblackgraph/reference/triangularization.py
import numpy as np

from dataclasses import dataclass
from typing import Dict, Sequence
from collections import defaultdict

from redblackgraph.reference.topological_sort import topological_sort

@dataclass
class Components:
    ids: Sequence[int]
    max_relationship: Sequence[int]
    size_map: Dict[int, int] # keyed by component id, valued by size of component

    def get_permutation_basis(self):
        # this yeilds a list of tuples where each tuple is the size of the component, the component id of␣
↪the vertex,
        # the max rel(u,v) for the vertex and the id of the vertex. We want the nodes
        # ordered by components size, component id, max rel(u,v), finally by vertex id
        return sorted(
            [(self.size_map[element[1][0]],) + element[1] + (element[0],)
             for element in enumerate(zip(self.ids, self.max_relationship))],
            reverse=True
        )

@dataclass
class Triangularization:
    A: Sequence[Sequence[int]]
    label_permutation: Sequence[int]

def find_components_extended(A: Sequence[Sequence[int]]) -> Components:
    """
    Given an input adjacency matrix (assumed to be transitively closed), find the distinct
    graph components
    :param A: input adjacency matrix
    :return: a tuple of:
      [0] - a vector matching length of A with the elements holding the connected component id of
      the identified connected components - labeled u
      [1] - a vector matching length of A with the elements holding the max n_p for the corresponding
      row - labeled v
      [2] - a dictionary keyed by component id and valued by size of component
    """
    n = len(A)
    u = [0] * n
    v = [0] * n
    q = defaultdict(lambda: 0)
    component_number = 1
    u[0] = component_number
    q[component_number] += 1
    for i in range(n):
        row_max = -2
        if u[i] == 0:
            component_number += 1
            u[i] = component_number
            q[component_number] += 1
        row_component_number = u[i]
        for j in range(n):
            if A[i][j] != 0:
                row_max = max(A[i][j], row_max)
                if u[j] == 0:
                    u[j] = row_component_number
                    q[row_component_number] += 1
                elif u[j] != row_component_number:
                    # There are a couple cases here. We implicitely assume a new row
                    # is a new component, so we need to back that out (iterate from 0
                    # to j), but we could also encounter a row that "merges" two
                    # components (need to sweep the entire u vector)
                    for k in range(n):
                        if u[k] == row_component_number:
                            u[k] = u[j]
                            q[row_component_number] -= 1
                            q[u[j]] += 1
                    component_number -= 1
                    row_component_number = u[j]
        v[i] = row_max
    return Components(u, v, {k:v for k,v in q.items() if v != 0})

def _get_triangularization_permutation_matrices(A):
    """
    u, v, and q are computed via find_components_extended, and then used to compute a
    permutation matrix, P, and P_transpose
    :param A:
    :return: the permutation matrices that will canonical_sort A
    """
    permutation_basis = find_components_extended(A).get_permutation_basis()

    # from the permutation basis, create the permutation matrix
    n = len(permutation_basis)
    P = np.zeros(shape=(n, n), dtype=np.int32)
    P_transpose = np.zeros(shape=(n, n), dtype=np.int32)
    # label_permutation can be calculated as P @ np.arrange(n), but since we are running the index do it␣
 ↪here
    label_permutation = np.arange(n)
    for idx, element in enumerate(permutation_basis):
        label_permutation[idx] = element[3]
        P[idx][element[3]] = 1
        P_transpose[element[3]][idx] = 1
    return P, P_transpose, label_permutation


def canonical_sort(A: Sequence[Sequence[int]]) -> Triangularization:
    """
    Canonically sort the matrix.

    This form of triangularization is canonical. Graph components will appear in adjacent
    rows starting with the largest component in rows 0-n, the next largest in n+1-m, etc.
    Should the graph hold components of the same size, the component id will be used to
    order one above the other. Within a component, row ordering is determined first by
    maximum relationship value in a row and finally by original vertex id.

    This is an expensive operation. First it assumes that A is transitively closed (O(n^3)).
    It then computes the components of the graph (O(n^3)). It then sorts the resultant
    component information (O(n logn)). Based on this it computes permutation matrices (O(n))
    and finally uses the permutation matrices to reorder the graph (O(n^2))

    :param A: input matrix (assumed to be transitively closed)
    :param P: the transposition matrices (P and P_transpose)
    :return: an upper triangular matrix that is symmetrical to A (a relabeling of the graph vertices)
    """

    P, P_t, label_permutation = _get_triangularization_permutation_matrices(A)

    # triagularize A
    return Triangularization((P @ A @ P_t), label_permutation)

def triangularize(A: Sequence[Sequence[int]]) -> Triangularization:
    """
    Relabel the graph so that the resultant Red Black adjacency matrix is upper triangular

    This form of triangularization is not canonical, it is only guaranteed to produce an upper
    triangular representation. It does so by topologically sorting the graph (O(V+E)). Then
    producing permutation matrices (O(n)). Finally using the permutation matrices to reorder the
    graph (O(n^2)).
    Whereas canonical_sort assumes that the input matrix is transitively closed,
    this version does not.
    :param A: input red black adjacency matrix
    :return: an upper triangular matrix that is symmetrical to A (a relabeling of the graph vertices)
    """

    # step 1: determine topological ordering of nodes in the graph
    n = len(A)
    ordering = topological_sort(A)
    # step 2: setup the permutation matrix
    P = np.zeros(shape=(n, n), dtype=np.int32)
    for idx, i in enumerate(ordering):
        P[idx][i] = 1
    # step 3: permute the matrix and return triangularization
    return Triangularization(P @ A @ P.T, ordering)
```

[ ]:
```
from redblackgraph.reference.triangularization import triangularize
triangularize(transitive_closure(R).W).A
```

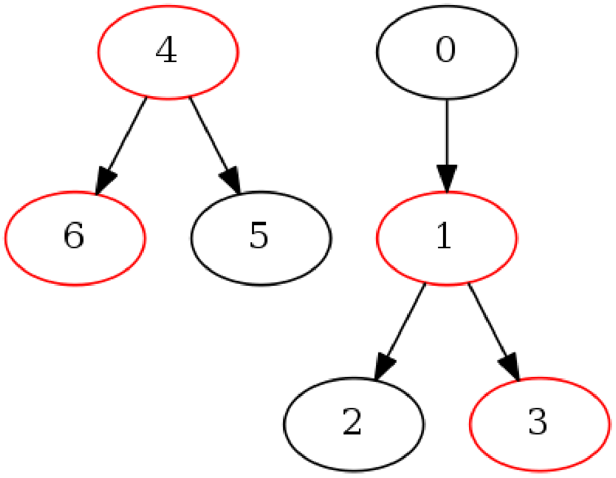

Figure 7: Graph with Components (Canonical Form)

5 Appendix A

5.1 Determinants

Let's explore the determinants of the class of matrices that represent Red Black Graphs. Staring with the simple case of a $2x2$ matrix.

$$|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - cb$$

As per formal definition, a and $d \in \{-1, 1\}$; b defines the relationship from the vertex represented by the first row to the vertex represented by the second row; c defines the relationship from the vertex represented by the second row to the vertex represented by the first row.

As per constraints (no cycles) if b is non-zero then c must be zero and conversely if c is non-zero, b must be zero. Therefore for a $2x2$ matrix, A, $det(A) \in \{-1, 1\}$.

Consider the case of a $3x3$ matrix.

$$|A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = aei + bfg + cdh - ceg - bdi - afh$$

As in the $2x2$ case, the product of the diagonals is constrained to $\{-1, 1\}$ and all other terms will be zero as they either represent the cycle of path length 2 or path length 1. Let's label the vertex representing by the first row as α, the second row as β and the third row as γ. Let's look at the bfg term. b represents the relationship from α to β, f represents the relationship from β to γ and g represents the relationship from γ to α. This term defines a cycle of path length 2 and at least one of the terms must be zero by constraint.

Let's look at the ceg term. c represents the relationship from α to γ, e represents the relationship from β to itself and g represents the relationship from γ to α. Again, by constraint, either c or g must be zero. Therfore ceg will be zero. Likewise, bdi and afh terms will be zero.

While the $2x2$ and $3x3$ cases are interesting, this line of reasoning doesn't extend to finding the determinant of higher dimensional matrices. As any Red Black graph can be represented in it's canonical form, an upper triangular matrix, we observe that: $\det R = \begin{cases} 1, & \text{if } |V_{red}| \text{ is even,} \\ -1, & \text{if } |V_{red}| \text{ is odd.} \end{cases}$

6 Appendix B

6.1 Implementation Notes on Numpy Extension

Following are some python examples. In addition to the simple implementation presented above in pure python, for performance optimized linear algebra operations, extension modules are provided by Numpy, SciPy, etc. The redblackgraph module also provides extension model implementations that are described below

6.2 rb.array / rb.matrix

The redblackgraph module provides two Numpy extensions, one for array and one for matrix.

The distinctive characteristics of these classes are matrix multiplication has been overridden to support the avos product, as well as methods defined for transitive_closure and relational_composition

To motivate the examples, let's model my familial relationshps. I'm (D) the child of Ewald (E) and Regina (R). Ewald and Marta (M) also have a child, my half-brother, Harald (H). Ewald's parents were Michael (Mi) and Amalie (A). Regenia's parents were John (J) and Inez (I). John also had a son Donald (Do) with Evelyn (Ev). Michael's parents were George (G) and Mariea (Ma). Finally, John's parents were Samuel (S) and Emeline (Em).

This set of relationships is represented by the graph below

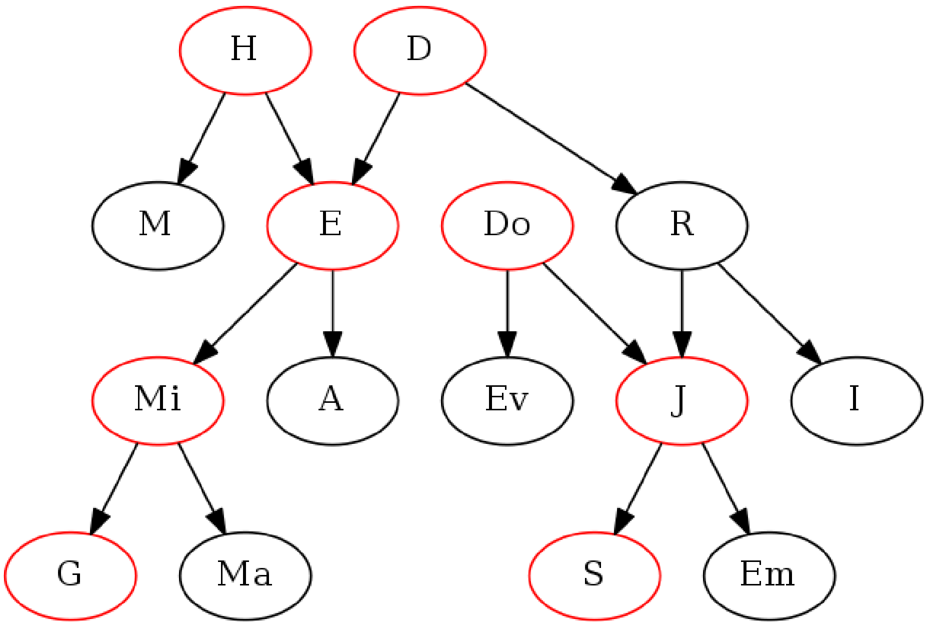

Figure 8: Graph for Exploring Python Implementation

We'll model this as a RedBlackGraph denoting each vertex numerically in the order introduced in the above narrative, e.g. D:0, E:1, R:2, M:3, H:4, Mi:5, A:6, J:7, I:8, Do:9, Ev:10, G:11, Ma:12, S:13, Em14

In these examples, we'll first calculate transitive closure then we'll remove the node (row/column) for John, create the simple row and column vectors for John and use a relational composition to recostruct a transitive closure equivalent. Finally we'll get some timings to compare implementations.

6.3 Simple Implementation

6.3.1 Transitive Closure

[ ]:
```
import numpy as np
import redblackgraph.reference as smp
import copy
#       D    E    R    M    H   Mi    A    J    I   Do   Ev    G   Ma    S   Em
A = [[-1,  2,  3,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0], # D
     [ 0, -1,  0,  0,  0,  2,  3,  0,  0,  0,  0,  0,  0,  0,  0], # E
     [ 0,  0,  1,  0,  0,  0,  0,  2,  3,  0,  0,  0,  0,  0,  0], # R
     [ 0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0], # M
     [ 0,  2,  0,  3, -1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0], # H
     [ 0,  0,  0,  0,  0, -1,  0,  0,  0,  0,  0,  2,  3,  0,  0], # Mi
     [ 0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0], # A
     [ 0,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0,  0,  0,  2,  3], # J
     [ 0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0], # I
     [ 0,  0,  0,  0,  0,  0,  0,  2,  0, -1,  3,  0,  0,  0,  0], # Do
     [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0], # Ev
     [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0], # G
     [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0], # Ma
     [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0], # S
     [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1]  # Em
    ]
B = copy.deepcopy(A)
res = smp.transitive_closure(B)
```

```
print(f"A_star:\n{res.W} \ndiameter: {res.diameter}")
```

6.3.2 Vertex Relational Composition

For illustrative purposes, let's remove John from the rb.array representation of the graph

[ ]:
```
#       D   E   R   M   H  Mi   A   I  Do  Ev   G  Ma   S  Em
A1 = [[-1,  2,  3,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0], # D
      [ 0, -1,  0,  0,  0,  2,  3,  0,  0,  0,  0,  0,  0,  0], # E
      [ 0,  0,  1,  0,  0,  0,  0,  3,  0,  0,  0,  0,  0,  0], # R
      [ 0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0], # M
      [ 0,  2,  0,  3, -1,  0,  0,  0,  0,  0,  0,  0,  0,  0], # H
      [ 0,  0,  0,  0,  0, -1,  0,  0,  0,  0,  2,  3,  0,  0], # Mi
      [ 0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0], # A
      [ 0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0], # I
      [ 0,  0,  0,  0,  0,  0,  0,  0, -1,  3,  0,  0,  0,  0], # Do
      [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0], # Ev
      [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0], # G
      [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0], # Ma
      [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0], # S
      [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1]  # Em
     ]
B1 = copy.deepcopy(A1)
res = smp.transitive_closure(B1)
print(f"A1_star:\n{res.W} \ndiameter: {res.diameter}")
```

Observation: I am no longer related to Samuel nor Emeline, but that the diameter is still 3 (my relationship to George and Mariea).

Let's look at the row (u) and column (v) vectors that would define John in relationship to A1 as well as the relational_composition of A1 with u and v.

[ ]:
```
#      D   E   R   M   H  Mi   A   I  Do  Ev   G  Ma   S  Em
u = [[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  2,  3]]
v = [[ 0], # D
     [ 0], # E
     [ 2], # R
     [ 0], # M
     [ 0], # H
     [ 0], # Mi
     [ 0], # A
     [ 0], # I
     [ 2], # Do
     [ 0], # Ev
     [ 0], # G
     [ 0], # Ma
     [ 0], # S
     [ 0], # Em
    ]
A_lambda = smp.vertex_relational_composition(u, A1, v, -1)
A_lambda
```

6.3.3 Edge Transitive Closure

Using the above example, remove the relationship from Regina to John

[ ]:
```
#        D   E   R   M   H  Mi   A   I  Do  Ev   G  Ma   S  Em   J
R1 = [[ -1,  2,  3,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],  # D
      [  0, -1,  0,  0,  0,  2,  3,  0,  0,  0,  0,  0,  0,  0,  0],  # E
      [  0,  0,  1,  0,  0,  0,  0,  3,  0,  0,  0,  0,  0,  0,  0],  # R
      [  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],  # M
      [  0,  2,  0,  3, -1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],  # H
      [  0,  0,  0,  0,  0, -1,  0,  0,  0,  0,  2,  3,  0,  0,  0],  # Mi
      [  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0],  # A
      [  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0],  # I
      [  0,  0,  0,  0,  0,  0,  0,  0, -1,  3,  0,  0,  0,  0,  2],  # Do
      [  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0],  # Ev
      [  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0,  0],  # G
      [  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0],  # Ma
      [  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0,  0],  # S
      [  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0], # Em
      [  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  2,  3, -1]  # J
      ]
R = smp.transitive_closure(R1).W
# Missing edge is R -> J, 2
A_lambda = smp.edge_relational_composition(R, 2, 14, 2)
A_lambda
```

6.3.4 Timings

[ ]:
```
%%timeit
B1 = copy.deepcopy(A1)
res = smp.transitive_closure(B1)
```

[ ]:
```
%%timeit
A_lambda = smp.vertex_relational_composition(u, A1, v, -1)
```

[ ]:
```
%%timeit
A_lambda = smp.edge_relational_composition(R, 2, 14, 2)
```

6.4 Optimized Implementation

6.4.1 Transitive Closure

[ ]:
```
import redblackgraph as rb
#                D   E   R   M   H  Mi   A   J   I  Do  Ev   G  Ma   S  Em
A = rb.array([[-1,  2,  3,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0], # D
              [ 0, -1,  0,  0,  0,  2,  3,  0,  0,  0,  0,  0,  0,  0,  0], # E
              [ 0,  0,  1,  0,  0,  0,  0,  2,  3,  0,  0,  0,  0,  0,  0], # R
              [ 0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0], # M
              [ 0,  2,  0,  3, -1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0], # H
              [ 0,  0,  0,  0,  0, -1,  0,  0,  0,  0,  0,  2,  3,  0,  0], # Mi
              [ 0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0], # A
              [ 0,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0,  0,  0,  2,  3], # J
              [ 0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0], # I
              [ 0,  0,  0,  0,  0,  0,  0,  2,  0, -1,  3,  0,  0,  0,  0], # Do
              [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0], # Ev
              [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0], # G
              [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0], # Ma
              [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0], # S
              [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1]  # Em
             ], dtype=np.int32)
```

[ ]:
```
result = A.transitive_closure()
print(f"A_star:\n{result.W} \ndiameter: {result.diameter}")
```

6.4.2 Vertex Relational Composition

For illustrative purposes, let’s remove John from the rb.array representation of the graph

[ ]:
```
#                 D   E   R   M   H  Mi   A   I  Do  Ev   G  Ma   S  Em
A1 = rb.array([[-1,  2,  3,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0], # D
               [ 0, -1,  0,  0,  0,  2,  3,  0,  0,  0,  0,  0,  0,  0], # E
               [ 0,  0,  1,  0,  0,  0,  0,  3,  0,  0,  0,  0,  0,  0], # R
               [ 0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0], # M
               [ 0,  2,  0,  3, -1,  0,  0,  0,  0,  0,  0,  0,  0,  0], # H
               [ 0,  0,  0,  0,  0, -1,  0,  0,  0,  0,  2,  3,  0,  0], # Mi
               [ 0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0], # A
               [ 0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0], # I
               [ 0,  0,  0,  0,  0,  0,  0,  0, -1,  3,  0,  0,  0,  0], # Do
               [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0], # Ev
               [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0], # G
               [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0], # Ma
               [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0], # S
               [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1]  # Em
              ], dtype=np.int32)
result = A1.transitive_closure()
print(f"A1_star:\n{result.W} \ndiameter: {result.diameter}")
A1_star = result.W
```

Observation: I am no longer related to Samuel nor Emeline, but that the diameter is still 3 (my relationship to George and Mariea).

Let's look at the row (u) and column (v) vectors t

[ ]:
```
#              D   E   R   M   H  Mi   A   I  Do  Ev   G  Ma   S  Em
u = rb.array([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  2,  3]], dtype=np.int32)
v = rb.array([[ 0],
              [ 0],
              [ 2],
              [ 0],
              [ 0],
              [ 0],
              [ 0],
              [ 0],
              [ 2],
              [ 0],
              [ 0],
              [ 0],
              [ 0],
              [ 0],
             ], dtype=np.int32)

u_lambda = u @ A1_star
v_lambda = A1_star @ v
print(f"u_lambda:\n{u_lambda}")
print(f"v_lambda:\n{v_lambda}")

A_lambda = A1_star.vertex_relational_composition(u, v, -1)
print(f"A_lambda:\n{A_lambda}")
```

6.4.3 Edge Transitive Closure

Using the above example, remove the relationship from Regina to John

[ ]:
```
#                D   E   R   M   H  Mi   A   I  Do  Ev   G  Ma   S  Em   J
R1 = rb.array([[ -1,  2,  3,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],  # D
               [  0, -1,  0,  0,  0,  2,  3,  0,  0,  0,  0,  0,  0,  0,  0],  # E
               [  0,  0,  1,  0,  0,  0,  0,  3,  0,  0,  0,  0,  0,  0,  0],  # R
               [  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],  # M
               [  0,  2,  0,  3, -1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],  # H
               [  0,  0,  0,  0,  0, -1,  0,  0,  0,  0,  2,  3,  0,  0,  0],  # Mi
               [  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0],  # A
               [  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0],  # I
               [  0,  0,  0,  0,  0,  0,  0,  0, -1,  3,  0,  0,  0,  0,  2],  # Do
               [  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0],  # Ev
               [  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0,  0],  # G
               [  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0],  # Ma
               [  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1,  0,  0],  # S
               [  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0],  # Em
               [  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  2,  3, -1]   # J
      ])
R = R1.transitive_closure().W
# Missing edge is R -> J, 2
A_lambda = R.edge_relational_composition(2, 14, 2)
```

```
A_lambda
```

6.4.4 Timings

[ ]:
```
%%timeit
result = A.transitive_closure()
A1_star = result.W
```

[ ]:
```
%%timeit
A_lambda = A1_star.vertex_relational_composition(u, v, -1)
```

[ ]:
```
%%timeit
A_lambda = R.edge_relational_composition(2, 14, 2)
```

6.5 Miscellaneous Linear Algebra

[ ]:
```
from numpy.linalg import det
det(A_lambda)
```

[ ]:
```
A_lambda.cardinality()
```

[ ]:
```

```