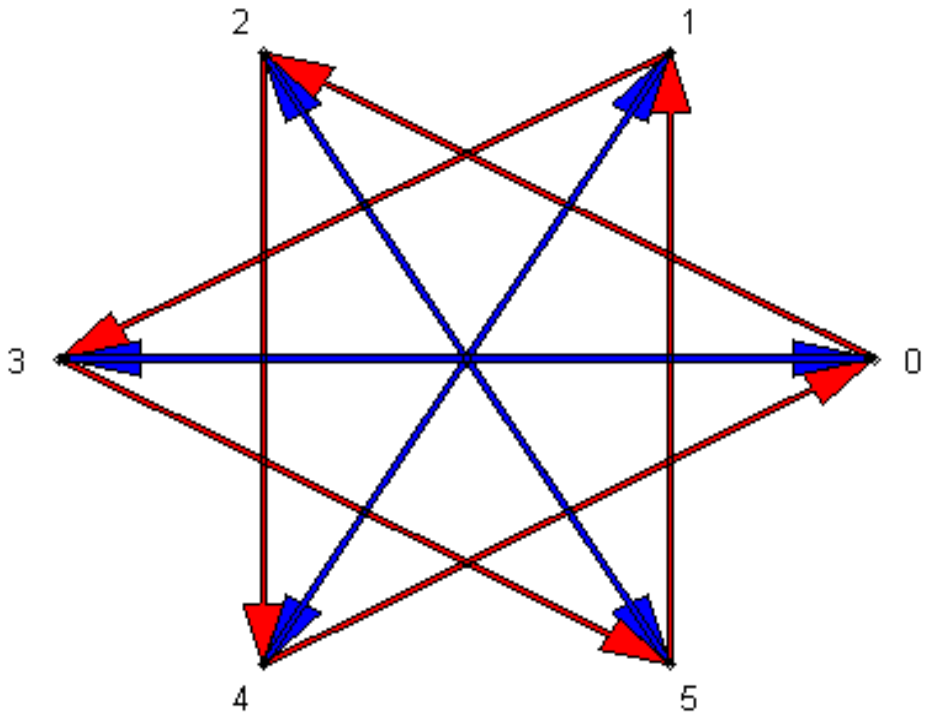
Hamiltonian Paths and Cayley Digraphs of Algebraic Groups.



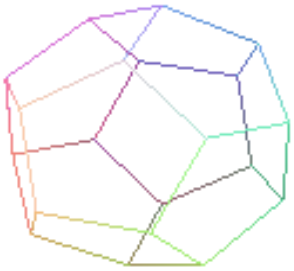
Sonja Willis

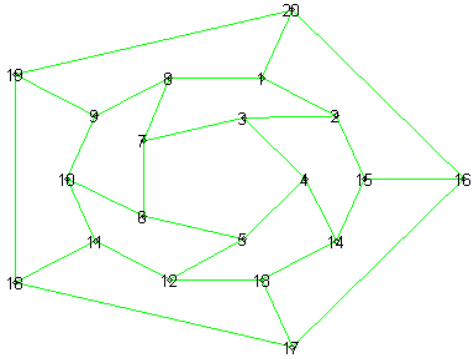
UCSD Honors Thesis 2001

Table of Contents

Introduction

The Irish Mathematician Sir William Hamilton labeled the twenty vertices of a regular dodecahedron with names of famous cities and asked various questions. The questions he asked surrounding the dodecahedron came to be known as the Around the World problem, because the dodecahedron looks like a sphere. One such question is does there exist a path from one city through every other city with out visiting two cities twice? Such a path is called a Hamiltonian path. The figure on the right is a digraph of a dodecahedron where each of the numbered vertices is a city. The vertices are numbered in a manner that if one follows them in numerical order, a Hamiltonian path is traced. This paper looks at the problem of finding Hamiltonian paths, not of dodecahedrons, but of Cayley digraphs of Algebraic groups.





This research project combines programming in Maple and mathematics, and focuses on the interplay between the two. The objective of this project is to develop an algorithm for finding Hamiltonian paths of graphs, such as the graph above on the right, and to use that algorithm to draw conclusions about Hamiltonian paths in the Cayley digraphs of Algebraic groups. Another goal of the project was to write a program in Maple that would draw a Cayley digraph in a manner that is useful and educational to a viewer. This paper is a guide to the process of writing programs and using them to do mathematics. First some background material will be covered, and the new terms specific to this project will be defined. Then the procedures that were written will be described in detail, with many example of their use. Finally the conclusions that were made about Algebraic groups will be discussed. The interplay between mathematics and programming was crucial to this project. With out the math, there would be no motivation for the programming, and without the programming, the plethora of examples that could be studied would have been diminished.

Background

Generating Set of a Group

To create Cayley digraphs, and to find Hamiltonian Paths of groups the notion of a generating set needs to be discussed. Maple denotes a set with curly brackets {}, and a list with square brackets []. To define a generating set let S be the set $\{ s_1 , s_2 ,..., s_n \}$ and let $\langle S \rangle$ be the smallest subgroup containing S. If $\langle S \rangle$ is equal to the group, then we say S is a set of generators for the group. $\langle S \rangle$ is the set of "words" in $\{ s_1 , s_2 ,..., s_n , s_1^{(-1)} s_2^{(-1)} ,..., s_n^{(-1)} \}$. If there exists $\langle S \rangle$ such that S only has one element then group is said to be cyclic. An example of this concept is illustrated below.[3]

Example

As an example examine the group Z_6 which is the group of integers (mod 6) with addition as a binary operation. Z_6 has six elements {0,1,2,3,4,5}, and there are several different generating sets. One generating set is {1}. Starting at the identity 0 and adding 1 recursively yields: $\{ 0 + 1 = 1,\ 1 + 1 = 2,\ 2 + 1 = 3,\ 3 + 1 = 4,\ 4 + 1 = 5,\ 5 + 1 = 6 = 0 \text{ (mod6)} \} = \{1 , 2 , 3 , 4 , 5 , 0 \}$, therefore {1} generates the whole group. Since there exists a generating set for Z_6 with only one element in it, then Z_6 is cyclic. Another generating set of Z_6 is {2,3}. All elements of Z_6 can be represented as a word in the generating set. For example $1 = 3 + 2 + 2 = 7 = 1 \text{ (mod6)}$, and $5 = 3 + 2$. All elements can be represented in such a manner. This group will be revisited after defining the Cayley digraph to give a more visual description of the generating set of a group.

Group Representations

When using Maple to do anything concerning group theory, the "group" package must be accessed. The reader can explore the package further by typing ?group on an execution line in a Maple worksheet (see [4]). The group package has two commands that can be used to input a group: permgroup and grelgroup. "permgroup" stands for a Permutation group which is the set of permutations of {1,2,...,n} that form a group under function composition. "grelgroup" stands for generators and relations, which is another way of defining a group. To define a group in terms of generators and relations one needs a set of elements that will generate the group and a set of equations (called relations) that specify the conditions that these generators are to satisfy. To illustrate these commands the group S_3, which is a set of permutations of {1,2,3}, and forms a group under permutation multiplication, will be used.

When using the command permgroup, the first argument is the degree of the group, and should be an integer. The second argument is a set of group generators. Each generator is represented in disjoint cycle notation. For S_3 the degree is 3 because it is the set of permutations of 1,2,3, and the set of generators used is {(12), (123)}.

```
> with(group):
> PG:=permgroup(3,{[[1,2]],[[1,2,3]]});
```

$$PG := \text{permgroup}(3, \{[[1,2]], [[1,2,3]]\})$$

In order to check to see that PG is the group S_3 the order and the elements can be printed. The order of S_3 is 6 because there are 6 ways to permute three numbers, and the elements are those six permutations.

```
> grouporder(PG); elements(PG);
```

$$6$$

$$\{[\,], [[1,2]], [[1,2,3]], [[1,3,2]], [[2,3]], [[1,3]]\}$$

The command grelgroup takes as a first argument a set of Maple names, which stand for the generators of the group. The second argument is a set of "words" in the generators. A "word" is a list of generators and/or inverses of generators representing a product. As seen below a and b are the generators and [a,a,a], [b,b], [a,b,a,b] are a set of relations for S_3. [a,a,a], [b,b], [a,b,a,b] means that $a^2 = e$, $b^2 = e$, and $abab = e$ where e is the identity element.

```
> GG:=grelgroup({a,b},{[a,a,a],[b,b],[a,b,a,b]});
```

$$GG := \text{grelgroup}(\{a, b\}, \{[a,a,a], [b,b], [a,b,a,b]\})$$

To check that GG is S_3 the order can be printed but Maple does not have a command to print the elements of a grelgroup.

```
> grouporder(GG);
```

$$6$$

Since Maple does not have a command to print the elements of a grelgroup a procedure to do this called eelements was written. When running this procedure the user must pass to it a group, either a permutation group or a group defined by generators and relations. The procedure prints the elements by finding the cosets of a subgroup of the group. To see the Maple code for this procedure see Appendix A, under eelements.

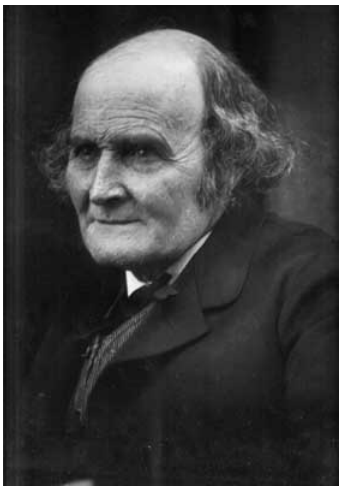
```
> eelements(GG);
```

$$\{[\,], [a,a,b], [a,b], [b], [a,a], [a]\}$$

This does not look like the elements of the first representation of S_3, however it will become clear after studying the Cayley digraph. This will be revisited later in the paper for clarification.

A procedure called MakePG was created that takes a set of generators and a set of relations and returns the corresponding permutation group. However for the set of generators and relations above it will not return S_3, it will return a subgroup of S_6 that is isomorphic to S_3. For the Maple code and a description of MakePG, see Appendix A.

Cayley Digraph of a Group



Arthur Cayley

> As for everything else, so for a mathematical theory: beauty can be perceived but not explained.
> -Arthur Cayley
> Quoted in J R Newman, *The World of Mathematics* (New York 1956). [8]

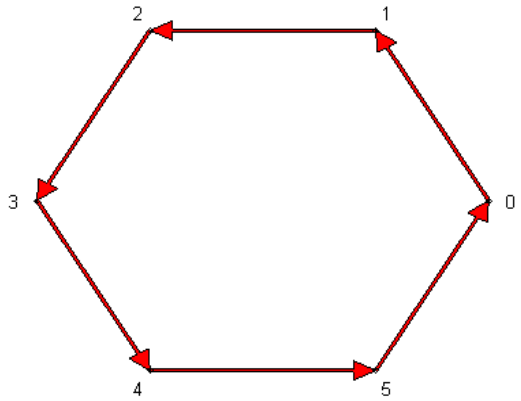
The Cayley digraph of a group provides a method of visualizing the group and it's properties. Properties such as commutativity, and the multiplication table of a group can be recovered from the Cayley digraph. A directed graph, or digraph is a finite set of points, called vertices, and a set of arrows, called arcs, connecting some of the vertices. The idea of representing a group in such a manner was originated by Cayley in 1878. [8]

Definition of the Cayley Digraph of a group

Let G be a finite group and S be a set of generators for G. The Cayley digraph of G with generating set S had two properties. The first property is that each element of G is a vertex of the Cayley digraph. The second property is, for a and b in G, there is an arc from a to b if and only if as = b for some s in S (see[2]). To give examples of Cayley digraphs the groups looked at previously will be revisited. The first example will be Z_6 with the generating set {1}, the second example will be the same group but with generating set {2,3}, and the final example will be S_3 with the generating set {(12),(123)}.

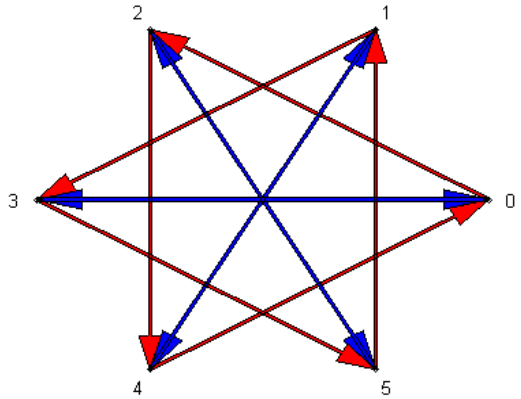
Examples

```
> Cgraph(Z6a);
```



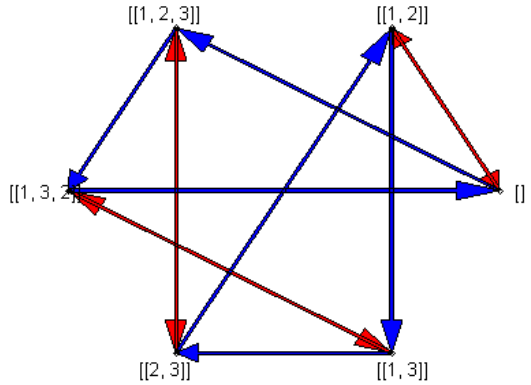
This is the Cayley digraph of Z_6 with generating set {1}. In the Cayley digraph each of the elements of the group Z_6 are the vertices of the digraph. The red arrow represents addition by 1, which is the only element in the generating set. The Cayley digraph illustrates several things about Z_6 with generating set {1}. The first point of interest is that 1 is of order 6, because if you start at the identity, 0, and add 1 six times which is equivalent to following the red arrow six times, you get back to the identity. Another property that should be noted is Z_6 is cyclic, because there is only one color of arrow, which implies that there exists a generating set with only one element.

> **Cgraph(Z6b);**



This is also a Cayley digraph for Z_6 but with the generating set {2,3}. The red arrow represents addition by 2, because starting at the identity 0, and following the red arrow yields 2. By the same logic, the blue arrow represents 3. The element 2 is of order 3 because starting at the identity and following the red arrow once yields 2, follow it again and you get to 4, follow it once more and you get back to 0, the identity, therefore applying the red arrow three times is equivalent to the identity. By the same principal, 3 has order 2. A quick way to see that an element in the generating set has order two is to look and see if it has a double-headed arrow, i.e. an arrow on both sides of the arc. Another property of the group that can be observed by looking at the Cayley digraph is commutativity. Start at any element of the group, say 2, and follow the blue arrow and then the red arrow, this results in 1. Now start at 2 again and follow the red arrow and then the blue arrow, this also yields 1. This shows that ab = ba. If this is true for all elements in the group then the group is commutative.

> `Cgraph(S);`



Above is the Cayley digraph for S_3, with generating set {(12),(123)}. The elements of the group are the vertices of the digraph and there are two elements in the generating set, (12) and (123). By looking at the identity element (0) = [[]], and following the red arrow, one can deduce that the red arrow represents multiplication by (12), and by the same logic, the blue arrow corresponds to multiplication by (123). The Cayley digraph illustrates several interesting facts about S_3. The Cayley digraph shows us that S_3 is a non-commutative group. This can be seen by starting at any element, say (13) and following the red arrow and then the blue arrow, which yields (0), then start at (13) again and follow the blue arrow and then the red arrow, this results in (123). Since (13) is different that (123) then S_3 is a non-commutative group.

The Multiplication table of the group can be recovered from the Cayley digraph. As previously stated, (12) corresponds to traveling the red arrow, therefore let (12) = R, and by the same logic, let (123) = B. Then using this R/B notation the rest of the elements can be represented in the same manner. (13) = RB, (132) = BB, and (23) = BR. Using this notation the multiplication table can be recovered by starting at the identity and traveling the corresponding arrows. For example (132)(13) = BBRB = (23) and (23)(132) = BRBB = (13). Below is the multiplication table for S_3. The multiplication table is dependent on the group, not on the generating set.

	0	$12 = R$	$123 = B$	$13 = RB$	$132 = BB$	$23 = BR$
0	0	12	123	13	132	23
$12 = R$	12	0	23	132	13	123
$123 = B$	123	13	132	23	0	12
$13 = RB$	13	123	12	0	23	132
$132 = BB$	132	23	0	12	123	13
$23 = BR$	23	132	13	123	12	0

As discussed previously in the representation section, S_3 can be represented as a grelgroup and the elements looked very different from permutations. As previous indicated the alternate representation of S_3 will be revisited in order to convince the reader that this representation adequately defines the group.

```
> GG:=grelgroup({a,b},{[a,a,a],[b,b],[a,b,a,b]});
```
$$GG := \text{grelgroup}(\{a, b\}, \{[a, a, a], [b, b], [a, b, a, b]\})$$

```
> eelements(GG);
```
$$\{[\ ], [a, a, b], [a, b], [b], [a, a], [a]\}$$

Looking again at the Cayley digraph and also at the multiplication table, the elements can be matched up with their corresponding permutations. Begin with element [a]. [a,a,a] is a relation which implies that $a^3 = e$ where e is the identity. Therefore a must be (123) because (123) is of order 3, so a corresponds to the blue arrow. Therefore b = (12), and corresponds to the red arrow and [ ] is obviously the identity element. [a,a] corresponds to two blue arrows, which if one looks at the multiplication table and the digraph, traveling two blue arrows yields (132), so [a,a] = (132). [a,b], by the same argument is a blue arrow followed by a red arrow which is (23), and lastly [a,a,b] is blue arrow, blue arrow, red arrow which yields (13). So one can see that the generators and relations given above do indeed define the group S_3 .

While writing procedures it was important to be able to draw Cayley digraphs and find Hamiltonian paths, given these two ways to represent groups. All of the procedures that were written accommodate for these two representations, and that is why the difference between them is discussed. There are important things to learn from each of the methods.

Drawing Cayley Digraphs in Maple

The definition of a Cayley digraph has been described, and examples of Cayley digraphs have been drawn in Maple, but the procedure for drawing them has not been explained. Maple does not have a procedure for drawing Cayley digraphs of groups. It does however have a package for drawing graphs. Further information about the draw package can be accessed by typing *?networks,draw* in the execution line of a Maple worksheet (see[5]). In order to draw a graph that represents a group, the first task at hand it to convert a group to a graph. A graph is a set of vertices and a set of vertex pairs. Each pair of vertices is specified as either a list or a set and is called an edge. Use of a list indicates direction from the tail to the head. To represent a group as a graph the objective is to have the elements of the group be the set of vertices, and to have a set of edges where an edge exists from one vertex to another if multiplying the first vertex by an element in the generating set yields the second vertex.

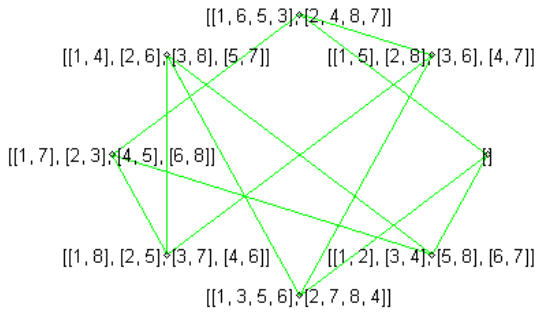
In order to convert a permutation group into a graph a procedure was created called DefGraph. The user passes this procedure the index of the permutation group and a generating set. Given this information it uses the procedure eelements to extract the elements of the group and then it multiplies these elements to the elements in the generating set to create the edges of the graph. The procedure then combines the set of elements, and the set of edges into a list that is now in the correct form for Maple to recognize it as a graph. Maple can display a graph using the draw command. D_4 will be used as an example. First convert the grelgroup to a permutation group using MakePG, then define the graph using DefGraph, and the draw the graph using Maples draw command.(Note that The "op" function extracts operands from an expression, so op(D4) just returns the 8, and the set of generators.)

```
> D4:=MakePG({a,b},{[a,a,a,a],[b,b],[a,b,a,b]});
```
$$D4 := \text{permgroup}(8, \{[[1, 3, 5, 6], [2, 7, 8, 4]], [[1, 2], [3, 4], [5, 8], [6, 7]]\})$$

```
> DD4:=DefGraph(op(D4));
```

DD4 := cay1

```
> draw (DD4);
```



As can be seen from this graph there are several problems. The first problem is that when a group gets large, the permutations get large, so it would be better to represent the edges with numbers rather than long permutations. The second problem is that the draw command draws green lines with no arrows. The graph would be closer to a Cayley digraph if arrows could be drawn at the end of the directed edges. The third problem is that there is no way of knowing which generator goes with which edges. Again the graph would look more like a Cayley digraph if the edges belonging to different elements in the generating set were different colors.

The first problem and part of the last problem were solved by using Groups32. Groups32 is a computer program which is based on the group tables for all groups of orders 1-32. It computes a variety of important properties of the groups. It also provides a "group theory" programming language. A program was written for Groups32 to provide data for all groups of orders 1-32 in a form that can be used by Maple. This program identifies generators for each group and produces a list which groups edges in the Cayley digraph by the generator used to produce them. This allows the production of colored Cayley digraphs for all groups up to order 32. It also allows application of the Hamiltonian Path procedure to all of these groups (see[7]).

The new representation given by Groups32 is a list. The first element in the list is a set of vertices, where the vertices are no longer permutations, but just numbers. The second element of the list is the number of generators, and the third element in the list is another list. The first element in this "sub list" is all the edges corresponding to the first element of the generating set, the second element is the edges corresponding to the second element of the generating set, and so on. Now the edges of the different elements of the generating set have been separated so that they can be drawn in different colors. Here is the new representation of D_4. It has edges 0-7, 2 elements in the generating set, and the list of the two sets, each set corresponding to the edges that belong to a separate element of the generating set.

```
> d4 := [{$0..7},  2 ,
            [{[0,1],[1,2],[2,3],[3,0],[4,5],[5,6],[6,7],[7,4]}
            ,{[0,4],[1,7],[2,6],[3,5],[4,0],[5,3],[6,2],[7,1]}]]:
```

In order to solve the problem of the graph having colors and arrows, Maple's draw

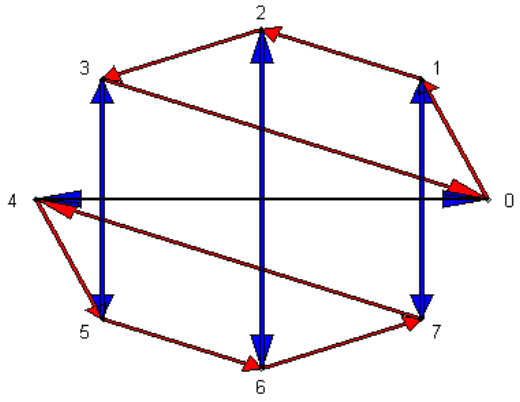
procedure needs modification. The directed edges need to be drawn with arrows, and there needs to be some way of passing color, edge width, arrow length and arrow width to the draw command. To view the modifications to the code of the draw command see Appendix A under Cgraph. A short overview of these modifications will be given.

A new procedure was created called xdraw that passed colors, length of the edge, and length and width of the arrow. Within xdraw, zdraw is called. zdraw is the modified draw command where it now recognizes these new arguments and incorporates them into drawing the graph. zdraw calls a subroutine named conc, which is a modification of the internal Maple command concentric. This had to be modified to recognize the new variables being passed. In order to use these modified programs the user only needs to know about four new variables. The first three are conc_a, conc_b, and conc_c. These three variables represent the line width, arrow width and arrow length, in that order. The user has the ability to change any of these specifications just by changing the assignment of the variable. The fourth variable is the set of colors. There are 25 different colors to choose from and to access a list the user can type ?*plot,color* in the execution line of a Maple worksheet. The procedure Cgraph is used to draw a Cayley digraph. Cgraph needs to be passed a graph in the notation that was discussed above with the group D_4. Cgraph uses this notation to make separate graphs with the same vertices. It makes as many graphs as there are generators. And each graph has one color and corresponds to a separate generator. Then all the graphs are displayed together to give the final result. Look at D_4 again, this time using Cgraph.

```
> conc_a:=.015:
  conc_b:=.1:
  conc_c:=.1:
> Colors := [red,blue,green,orange,yellow];
```

$$Colors := [red, blue, green, orange, yellow]$$

```
> Cgraph := proc(Cay_struc) local X,Vert,Edg,numgens,i;
                            global Colors;
            Vert := Cay_struc[1];
            numgens := Cay_struc[2];
            Edg := Cay_struc[3];
            for i from 1 to numgens do
                X[i] := xdraw(graph(Vert,Edg[i]),Colors[i]) od;
            display(convert(X,set))  end:
> Cgraph (d4);
```



One element of the generating set is red and of order 4 because following the red arrow four times returns to the identity. Call this generator 1, and the other element of the generating set, the blue arrow is of order 2, call this 4. Note that D_4 is a non-commutative group. The multiplication table for D_4 can be recovered by noticing that 1 = R, 2 = RR, 3 = RRR, 4 = B, 5 = BR, 6 = BRR, 7 = RB, where R corresponds to the red arrow and B corresponds to the blue arrow.

Cayley digraphs are more than just a visual representation of a group. They illustrate various properties of the group such a being cyclic, and commutativity. The multiplication table of the group can be recovered from these illustrations.

Hamiltonian Path

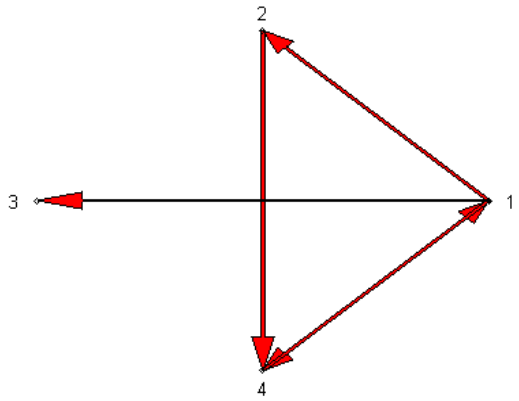
Sir William Hamilton

> On earth there is nothing great but man; in man there is nothing great but mind.
> -Sir William Rowan Hamilton
> *Lectures on Metaphysics.*[9]

The questions that Hamilton had when he invented his Around the World puzzle can be applied to Cayley digraphs. That is, one starts at some vertex and attempts to traverse the digraph by moving along arcs in such a way that each vertex is visited once. Such a sequence of arcs that passes through each vertex exactly once without returning to the starting point is called a Hamiltonian path, after Sir William Hamilton. One can find, or not find, a Hamiltonian path in any graph, but paths in Cayley digraphs will be discussed in this paper.

A procedure was created that determines whether or not a graph has a Hamiltonian path. If a graph has a Hamiltonian path then the procedure will return that path. If the graph does not have a Hamiltonian path then the procedure will return the statement "NO PATH, TRY ANOTHER VERTEX", and the last path that it could find. The main procedure written is called HamiltonainPath. It calls two other procedures, continuepath and backtrack. An example graph will be used to explain how all three of these procedures work. The example graph is not the graph of a group, and will be called SG. Below is a picture of SG. (Note that SGr is the sample graph in the notation that Cgraph needs to draw a Cayley digraph.)

```
> SGr := [{$1..4},1,[{[1,2],[2,4],[1,3],[1,4],[4,1]}]] :
> SG := graph(op(1,SGr),op(op(3,SGr))):
> Cgraph (SGr);
```



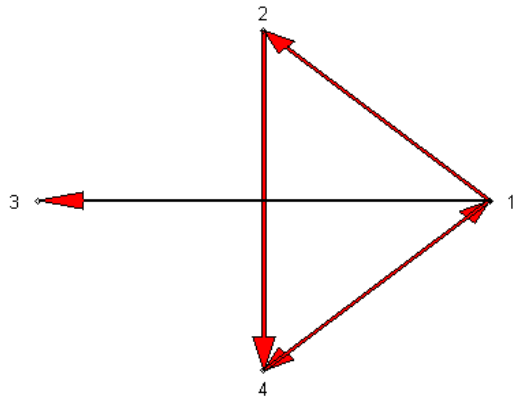
HamiltonianPath is the procedure written to find Hamiltonian paths of graphs. HamiltonianPath calls two subroutines:continuepath, and backtrack. An overview will be given to explain what the three procedures do and how they work together, and then each program will be defined and described in detail. SG, the sample graph, will be used to illustrate what is going on in the procedures.

For a quick overview, the user passes HamiltonianPath a graph and a starting vertex. HamiltonianPath passes that vertex to the subroutine continuepath as a path (a path is a sequence of vertices), where it tries to continue the path. This is repeated until a dead end is hit. Once a dead end is hit, HamiltonianPath passes the path to backtrack where it takes the last vertex off the path and labels it as "bad". From there either the path gets sent to continuepath, or stays in backtrack until it can go to continuepath. This backing up and going forward method is repeated until one of two things happens; one, the path has the same number of elements at the graph has vertices, or two, backtrack has backed up all the way to the beginning and all options have been exhausted. In the first case HamiltonianPath will print the path, in the second case it will print "NO PATH TRY ANOTHER VERTEX" and will return the last path it found.

Now each of the procedures will be described in more detail, using SG, the sample graph to aid in description. It is worth noting that print statements have been added to the procedures so that it will be easier to understand what is happening throughout the description. The procedures in Appendix A do not have print statements. Print statements slow down an algorithm considerably.

The continuepath Procedure

```
> continuepath := proc(Graph,path)
  local daugh;
print(`continuepath was passed`,path);
daugh:=sort([op(departures(path[1],Graph) minus {op(path)})]);
    if daugh = []
      then RETURN ([path,false]);
      else RETURN ([[daugh[1],op(path)],true]);
    fi;
end:
```



Continuepath is passed a graph, such as SG. It then finds the departures of the last element of the path. The departures of a vertex is a set of vertices which are at the head of the edges directed out of that vertex. For example the departures of 1 in the sample graph are:

```
> departures(1,SG);
```

{2, 3, 4}

The departures of 4 , and the departures of 3 are below.

```
> departures(4,SG); departures (3,SG);
```

{1}

{ }

Once continuepath finds the departures, it then does one of two things. If the set of departures in empty, like it was for the vertex 3, then continuepath will return the a list where the first element is the path, and the second element is the word false. If the set of departures is not empty then continuepath will also return a list. The first element of this list will be a new path that is made up of all the elements of the old path, with the first element of the departure set attached to the path, and the second element of the list will be the word true. For example, execute continuepath twice, the first time with 1 as the path and the second time with 3 as the path.

```
> continuepath(SG,[1]);
continuepath(SG,[3]);
```

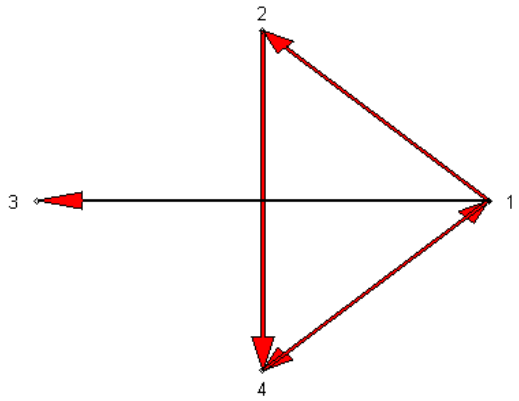
continuepath was passed, [1]

[[2, 1], *true*]

continuepath was passed, [3]

[[3], *false*]

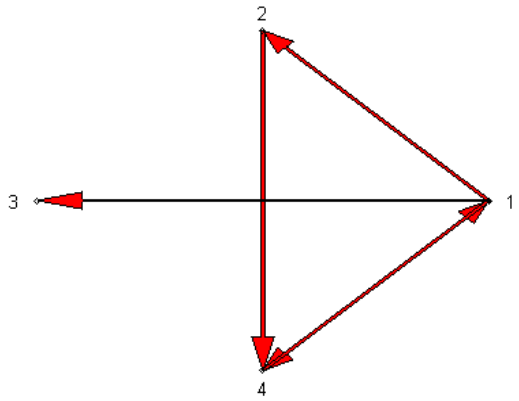
As can be seen above, continuepath continues on from 1 to 2, and returns a true, but in the case where the path is 3, and 3 has no departures, continuepath returns the path and a false. Continuepath is the procedure that adds vertices to the path.



The backtrack Procedure

```
> backtrack := proc( Graph, origpath)
  local bad, daugh, position, path ;
  path := origpath;
  do
    bad:=path [1];
    path:=path [2..nops(path)];
    if path = []
      then RETURN ([path, false]);
    fi;
 print (`after bad is off the path is`,path);
    daugh:=sort([op(departures(path[1],Graph) minus
{op(path)})]);
    member (bad,daugh,'position');
print(`the daughters of`, path[1], `are`,daugh,`the bad element is`, bad);
    if position < nops(daugh)
      then RETURN ([[daugh[position + 1],op(path)],true]);
    fi;
   od;
end:
```

If continuepath is the procedure that adds vertices to the graph, then backtrack is the procedure that takes those vertices off the path. No matter what path is passes to backtrack it will take off the first element in the path, and mark it as bad. Then it will do what continuepath does and find the departures. If the set of departures is not empty then it will put the last element of the departures on the list and will return a list with the new path and the word true. If the set of departures is empty, it will continue to backtrack and label elements as bad until it finds a set of departures that is not empty. If it backs all the way up until there is nothing left in the path, then it will return the empty path, and a false. For example look at the path 1, 3, which is represented in a list [3,1]. (Note that in the procedures a path starts from the right and goes to the left. This may seem backwards, but backwards is relative.)




```
> backtrack(SG,[3,1]);
```

after bad is off the path is, [1]

the daughters of, 1, *are*, [2, 3, 4], *the bad element is*, 3

[[4, 1], *true*]

Backtrack took the 3 off the list, went back to 1, found the departures, and went to 4 which is the last element in the set of departures, and returned a true. In order to see backtrack work it's magic look at the path from 2 to 4 to 1 to 3. If this path is sent to backtrack, it will have to back all the way up to the beginning because the set of departures will always have only the bad elements in it.

```
> backtrack (SG,[3,1,4,2]);
```

after bad is off the path is, [1, 4, 2]

the daughters of, 1, *are*, [3], *the bad element is*, 3

after bad is off the path is, [4, 2]

the daughters of, 4, *are*, [1], *the bad element is*, 1

after bad is off the path is, [2]

the daughters of, 2, *are*, [4], *the bad element is*, 4

[[ ], *false*]

A seen in the above example, backtrack will continue to back up until it can not back up anymore, and when that happens it returns an empty path and a false. Traveling forward is done with continuepath and backing up is done with backtrack. The procedure HamiltonianPath uses these to subroutines to find paths in graphs.

The HamiltonianPath Procedure

```
> HamiltonianPath := proc(Graph,start)
  local path, result;
  path := [start];
  do
    do
      result := continuepath (Graph,path);
```

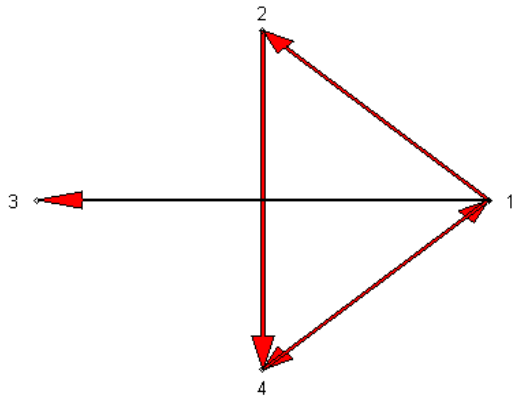
```
      path := result [1];

      if result [2] = false
        then break;
      fi;
    od ;

    if nops(result [1]) = nops(vertices(Graph))
      then RETURN (path);
    fi ;

    result:=backtrack (Graph,path);
    if result[2]=false
      then print (`NO PATH TRY ANOTHER VERTEX`); break ;
      else path :=result [1];
    fi;
  od;
end:
```

The HamiltonianPath procedure takes in a graph and a starting point. It makes the starting point the first element of the path and it sends the path to continuepath. As long as continuepath returns a true at the end of the list containing the path then HamiltonianPath continues to send the path to continuepath. When continuepath returns a false at the end of the list that contains the path, then HamiltonianPath sends the list to backtrack. Backtrack returns a list with true or false as the last element. If a true is sent, then the path is sent to continuepath again, if a false is passed, then no path exists and HamiltonianPath returns a message and the last path found before backtrack was called. Every time a vertex is added to the path, HamiltonianPath checks to see if the path has as many elements as the graph has vertices. If at any time the two have the same number of elements, then HamiltonainPath ends and returns the path that is found.




```
> HamiltonianPath(SG,1);
```

continuepath was passed, [1]

continuepath was passed, [2, 1]

continuepath was passed, [4, 2, 1]

after bad is off the path is, [2, 1]

the daughters of, 2, *are*, [4], *the bad element is*, 4

after bad is off the path is, [1]

the daughters of, 1, *are*, [2, 3, 4], *the bad element is*, 2

continuepath was passed, [3, 1]

after bad is off the path is, [1]

the daughters of, 1, *are*, [2, 3, 4], *the bad element is*, 3

continuepath was passed, [4, 1]

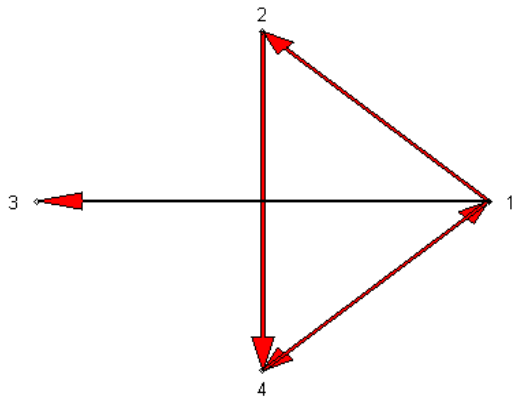
after bad is off the path is, [1]

the daughters of, 1, *are*, [2, 3, 4], *the bad element is*, 4

NO PATH TRY ANOTHER VERTEX

[4, 1]

The print statements along with the digraph can be followed to chart the progress of the procedure. This verifies that HamiltonianPath works as described. HamiltonianPath verifies that there does not exist a path from 1 in the sample graph. However a path does exist from vertex number 2. The path can be found without backtracking.




```
> HamiltonianPath(SG,2);
```

continuepath was passed, [2]

continuepath was passed, [4, 2]

continuepath was passed, [1, 4, 2]

[3, 1, 4, 2]

The subroutines created to make things easier for the user to find Hamiltonian paths of groups and graphs will now be described. The user to be able to find Hamiltonian paths by inputting any one of four forms of input:

❶ A graph and a starting vertex

❷ A permutation group

❸ A group defined by generators and relations

❹ A group in the notation obtained from Groups32, where there is a set of vertices, the number of elements in the generating set, and a list where the edges are separated into sets corresponding to the different elements of the generating set.

The procedure HamiltonianPath was created for the first form of input. Here is an example:

```
> A := graph({$1..5},{[1,2],[2,4],[1,3],[1,4],[4,1],[3,5]}) :
> HamiltonianPath(A,1);
```

NO PATH TRY ANOTHER VERTEX

[4, 1]

For the second and third form of input the procedure HamPath was created. The user can type either form in after the command and it will recognize which form is given. Here is an example:

```
> HamPath(3,{[[1,2]],[[1,2,3]]});
```

[[[1, 3, 2]], [[1, 2, 3]], [[2, 3]], [[1, 3]], [[1, 2]], [ ]]

```
> HamPath({a,b},{[a,a,a],[b,b],[a,b,a,b]});
```

[[[1, 4], [2, 5], [3, 6]], [[1, 5], [2, 6], [3, 4]], [[1, 6], [2, 4], [3, 5]],
[[1, 3, 2], [4, 5, 6]], [[1, 2, 3], [4, 6, 5]], [ ]]

Hpath was created for the forth form of input. Here is an example:

```
> Z:=[{0,1,2,3,4,5},1,[{[0,1],[1,2],[2,3],[3,4],[4,5],[5,0]}]];
```

$Z := [\{0, 1, 2, 3, 4, 5\}, 1, [\{[0, 1], [4, 5], [5, 0], [3, 4], [1, 2], [2, 3]\}]]$

```
> Hpath(Z);
```

[5, 4, 3, 2, 1, 0]

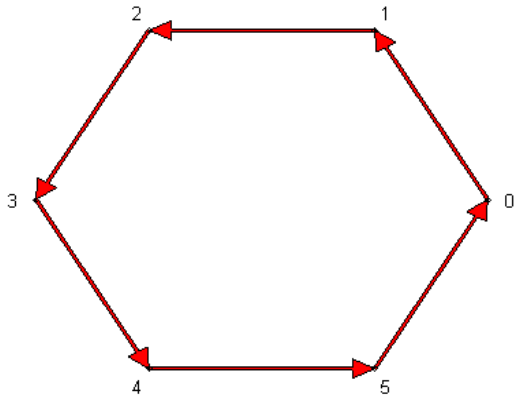
Do All groups have Hamiltonian Paths?

After creating a procedure for finding Hamiltonian paths the next task was to use the program to determine which groups with which generating sets have Hamiltonian paths? Seven cases were tested to see if paths were abundant or hard to find. These cases were Z_6, S_3, Z_8, Z_4 x Z_2, Z_2 x Z_2 x Z_2, D_4, and Q_8. The first two groups are of order 6 and the last five are of order 8. Of the two groups of order 6, Z_6 is Abelian and S_3 is not Abelian. Of the five groups of order 8, three are Abelian: Z_8, Z_4 x Z_2, Z_2 x Z_2 x Z_2 and two are non-Abelian D_4 (the dihedral group) and Q_8 (the group of quaternionic units).

The first group tested is Z_6. It is a cyclic group of order 6 and the generating set is $\{1\}$. It has a Hamiltonian path, which is printed in blue.

```
> CCay[7] := [{$0..5},1,[{[0,1],[1,2],[2,3],[3,4],[4,5],[5,0]}]]
:
Hpath(CCay[7]);
Cgraph(CCay[7]);
```

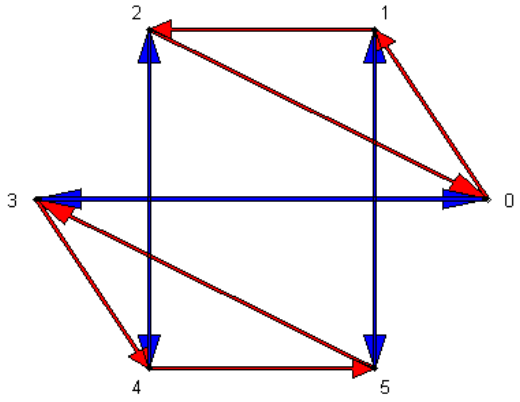
[5, 4, 3, 2, 1, 0]



Below is S_3 with the generating set {(12),(123)}. This Cayley digraph was displayed previously, and the blue line is a Hamiltonian path.

```
> CCay[8] := [{$0..5},2,[\
         {[0,1],[1,2],[2,0],[3,4],[4,5],[5,3]}\
        ,{[0,3],[1,5],[2,4],[3,0],[4,2],[5,1]}]] :
Hpath(CCay[8]);
Cgraph(CCay[8]);
```

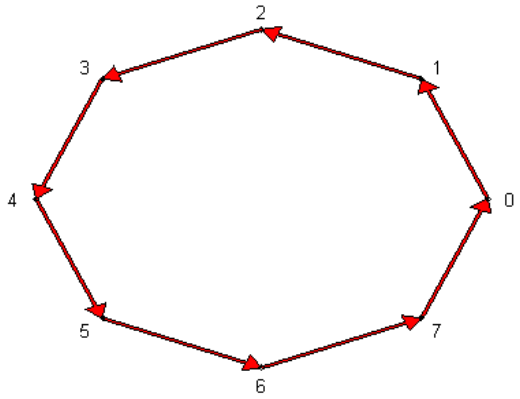
[3, 5, 4, 2, 1, 0]



Below is Z_8 . It is a cyclic group of order 8, and it obviously has a Hamiltonian path.

```
> CCay[10] := [{$0..7},1,[\
         {[0,1],[1,2],[2,3],[3,4],[4,5],[5,6],[6,7]\
        ,[7,0]}]] :
Hpath(CCay[10]);
Cgraph(CCay[10]);
```

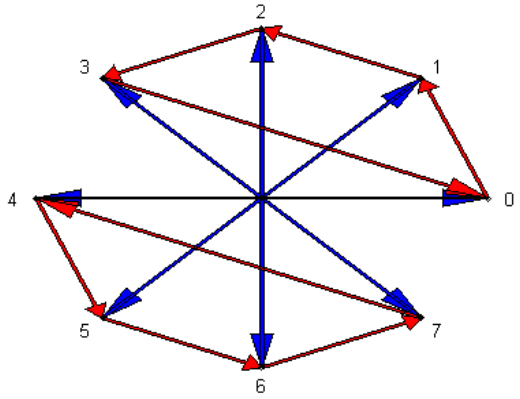
[7, 6, 5, 4, 3, 2, 1, 0]



The next group is Z_4 x Z_2 and it also has a Hamiltonian path. The red generator is of order 4 and the blue is of order 2. The two generators commute with one another and therefore the group is commutative.

```
> CCay[11] := [{$0..7},2,[
         {[0,1],[1,2],[2,3],[3,0],[4,5],[5,6],[6,7]\
        ,[7,4]},{[0,4],[1,5],[2,6],[3,7],[4,0],[5,1],[6,2]\
        ,[7,3]}]] :
Hpath(CCay[11]);
Cgraph(CCay[11]);
```

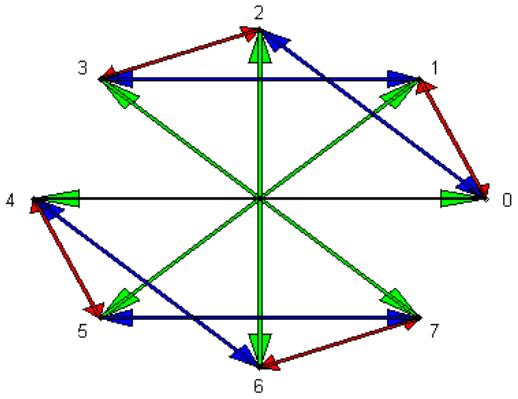
[6, 5, 4, 7, 3, 2, 1, 0]



Below is the group Z_2 x Z_2 x Z_2 and it requires three generators each of order 2. It also has a Hamiltonian path.

```
> CCay[12] := [{$0..7},3,[\
         {[0,1],[1,0],[2,3],[3,2],[4,5],[5,4],[6,7]\
        ,[7,6]},{[0,2],[1,3],[2,0],[3,1],[4,6],[5,7],[6,4]\
        ,[7,5]},{[0,4],[1,5],[2,6],[3,7],[4,0],[5,1],[6,2]\
        ,[7,3]}]] :
Hpath(CCay[12]);
Cgraph(CCay[12]);
```

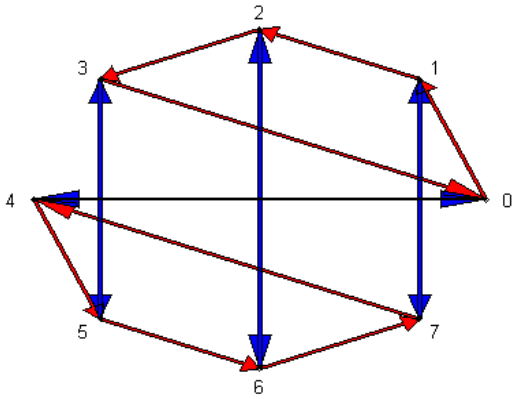
[7, 5, 4, 6, 2, 3, 1, 0]



Below is D_4 which is the group of symmetries of the square (the dihedral group), and it also has a Hamiltonian path. The red generator is of order 4 and the blue of order 2. They do not commute with each other.

```
> CCay[13] := [{$0..7},2,[\
        {[0,1],[1,2],[2,3],[3,0],[4,5],[5,6],[6,7]\
       ,[7,4]},{[0,4],[1,7],[2,6],[3,5],[4,0],[5,3],[6,2]\
       ,[7,1]}]] :
Hpath(CCay[13]);
Cgraph(CCay[13]);
```

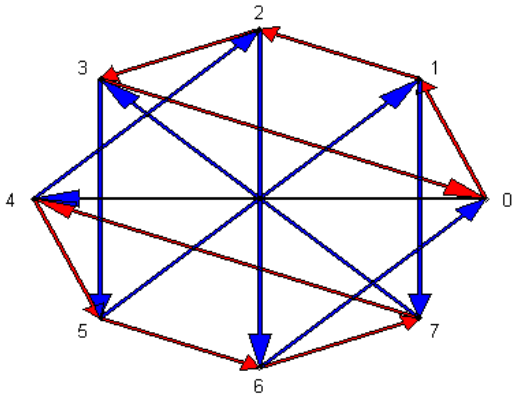
[4, 7, 6, 5, 3, 2, 1, 0]



The last group is Q_8. This group has two elements in the generating set, each of order 4, which do not commute. The quaternion group has elements ±1, ±i, ±j, ±k. The multiplication is like that for the cross product, ij=k, etc., however $i^2 = j^2 = k^2 = -1$. Generators are i and j. Vertex 0 is "1" and vertex 2 is -1. It has a Hamiltonian path also.

```
> CCay[14] :=
[{$0..7},2,[{[0,1],[1,2],[2,3],[3,0],[4,5],[5,6],[6,7]\
       ,[7,4]},{[0,4],[1,7],[2,6],[3,5],[4,2],[5,1],[6,0]\
       ,[7,3]} ]] :
Hpath(CCay[14]);
Cgraph(CCay[14]);
```

[4, 7, 6, 5, 3, 2, 1, 0]



All seven examples with minimal generating sets had Hamiltonian paths. Do all groups have Hamiltonian paths? Research on the subject yielded a theorem by Joseph Gallian. He proved that Abelian Groups have Hamiltonian paths.

Theorem: Let G be a finite Abelian group, and let S be any non-empty generating set for G. Then the Cayley digraph of G with generating set S has a Hamiltonian path.

The theorem is proved by induction on the number of elements in the generating set, and can be found in reference [2] on page 512. Gallian states that there are some Cayley digraphs for non-Abelian groups that do not have Hamiltonian paths, but he does not discuss the matter further. In order to find such a group, more groups and generating sets are needed. Using Groups32, minimal generating sets for all groups of order 1-32 were obtained (see [7]). There are 144 groups with order less than or equal to 32. All 144 of these groups with their minimal generating were run through a procedure that returns 5 things. The first is the group number (1-144), the second is a true or false, which corresponds to having or not having a Hamiltonian path. The third piece of information returned is the order of the group, the fourth is the number of elements in the generating set, and the fifth is how long it took to find the path. Here is the output of that procedure.

2, *true*, 2, 1, .005	12, *true*, 8, 3, .035	22, *true*, 12, 2, .095	32, *true*, 16, 3, .150
3, *true*, 3, 1, .005	13, *true*, 8, 2, .020	23, *true*, 12, 2, .036	33, *true*, 16, 4, .086
4, *true*, 4, 1, .010	14, *true*, 8, 2, .030	24, *true*, 12, 2, .120	34, *true*, 16, 3, .145
5, *true*, 4, 2, .014	15, *true*, 9, 1, .015	25, *true*, 13, 1, .025	35, *true*, 16, 3, .090
6, *true*, 5, 1, .010	16, *true*, 9, 2, .030	26, *true*, 14, 1, .081	36, *true*, 16, 3, .085
7, *true*, 6, 1, .015	17, *true*, 10, 1, .070	27, *true*, 14, 2, .045	37, *true*, 16, 2, .150
8, *true*, 6, 2, .074	18, *true*, 10, 2, .030	28, *true*, 15, 1, .031	38, *true*, 16, 2, .139
9, *true*, 7, 1, .015	19, *true*, 11, 1, .019	29, *true*, 16, 1, .030	39, *true*, 16, 2, .105
10, *true*, 8, 1, .015	20, *true*, 12, 1, .025	30, *true*, 16, 2, .045	40, *true*, 16, 2, .050
11, *true*, 8, 2, .025	21, *true*, 12, 2, .145	31, *true*, 16, 2, .320	41, *true*, 16, 2, .045

42, *true*, 16, 2, .050
43, *true*, 17, 1, .030
44, *true*, 18, 2, .455
45, *true*, 18, 1, .035
46, *true*, 18, 2, .305
47, *true*, 18, 2, .054
48, *true*, 18, 3, .179
49, *true*, 19, 1, .040
50, *true*, 20, 2, .314
51, *true*, 20, 1, .040
52, *true*, 20, 2, .056
53, *true*, 20, 2, 1.405
54, *true*, 20, 2, .060
55, *true*, 21, 1, .100
56, *true*, 21, 2, .600
57, *true*, 22, 1, .040
58, *true*, 22, 2, .065
59, *true*, 23, 1, .045
60, *true*, 24, 3, 6.141
61, *true*, 24, 2, 4.695
62, *true*, 24, 1, .045
☆
63, *false*, 24, 2, 1.804
☆
64, *true*, 24, 3, 5.420
65, *true*, 24, 2, .070
66, *true*, 24, 2, 4.873
67, *true*, 24, 2, .131
68, *true*, 24, 2, .180
69, *true*, 24, 2, .130
70, *true*, 24, 2, .179
71, *true*, 24, 2, .135
72, *true*, 24, 2, .070
73, *true*, 24, 2, 3.580
74, *true*, 24, 2, .070
75, *true*, 25, 2, .140
76, *true*, 25, 1, .045
77, *true*, 26, 1, .049
78, *true*, 26, 2, .080
79, *true*, 27, 3, .179
80, *true*, 27, 2, .080
81, *true*, 27, 1, .110
82, *true*, 27, 2, .095
83, *true*, 27, 2, 2.595
84, *true*, 28, 2, 1.061
85, *true*, 28, 1, .055
86, *true*, 28, 2, .150
87, *true*, 28, 2, .085
88, *true*, 29, 1, .110
89, *true*, 30, 1, .055
90, *true*, 30, 2, .121
91, *true*, 30, 2, 3.375
92, *true*, 30, 2, .085
93, *true*, 31, 1, .060
94, *true*, 32, 1, .119
95, *true*, 32, 2, 7.624
96, *true*, 32, 2, 1.930
97, *true*, 32, 2, 2.188
98, *true*, 32, 2, 1.470
99, *true*, 32, 2, 1.138
100, *true*, 32, 2, .995
101, *true*, 32, 2, 14.036
102, *true*, 32, 2, 1.570
103, *true*, 32, 2, 27.482
104, *true*, 32, 2, .730
105, *true*, 32, 2, 8.451
106, *true*, 32, 2, 5.490
107, *true*, 32, 2, 17.075
108, *true*, 32, 2, 49.727
109, *true*, 32, 2, 3.166
110, *true*, 32, 2, 1.750
111, *true*, 32, 2, .085
112, *true*, 32, 2, 4.020
113, *true*, 32, 2, 19.213
114, *true*, 32, 3, 80.652
115, *true*, 32, 3, 6.236
116, *true*, 32, 3, 54.755
117, *true*, 32, 3, 26.670
118, *true*, 32, 3, 48.210
119, *true*, 32, 3, 36.441
120, *true*, 32, 3, .261
121, *true*, 32, 3, .240
122, *true*, 32, 3, 11.400
123, *true*, 32, 3, 1.295
124, *true*, 32, 3, 64.603
125, *true*, 32, 3, 3.150
126, *true*, 32, 3, 2.100
127, *true*, 32, 3, 1.415
128, *true*, 32, 3, 118.045
129, *true*, 32, 3, .680
130, *true*, 32, 3, 12.610
131, *true*, 32, 3, 4.440
132, *true*, 32, 3, .185
133, *true*, 32, 3, 35.546
134, *true*, 32, 3, 15.385
135, *true*, 32, 3, 3.321
136, *true*, 32, 3, .120
137, *true*, 32, 3, 8.821
138, *true*, 32, 4, .274
139, *true*, 32, 4, .215
140, *true*, 32, 4, 1.465
141, *true*, 32, 4, .874
142, *true*, 32, 4, .154
143, *true*, 32, 4, .271
144, *true*, 32, 5, .260

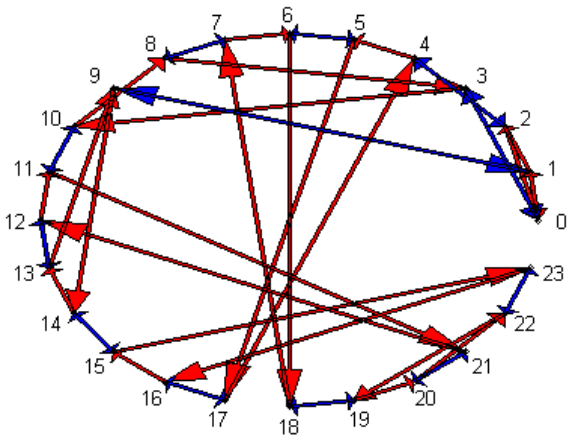
Running all groups of order 32 or less through this procedure returned only one group without a Hamiltonian path. This group is separated with red stars in the above data. The group is group number 63 and it has 24 elements and two elements in its generating set. This group

deserves further exploration.

Z_2 x A_4 : Three different generating sets

Out of all the groups that were looked at during this research project, group number 63 was the only group found not to have a Hamiltonian path. What does group number 63 look like?

```
> Cgraph(CCay[63]);
```



Group number 63 has two elements in it's generating set, one of order two and one of order three. Holsztynski and Strube talk about a group defined by generators and relations that doesn't have a path. They state G = {{a,b},$a^2 = b^3 = (aba\ b^2)^2 = e$} does not have a Hamiltonian path (see [3]). Since G has two elements in the generating set, one of order two and one of order three, one can ask if group 63 and G are equal. If G = group 63 then group 63 would have to have the relation $(aba\ b^2)^2 = e$. Looking at the Cayley digraph one can verify this relation. Although it is visually difficult, it is true. Therefore group 63 is the same as G and G is isomorphic Z_2 x A_4. It remains to verity that G does not have a Hamiltonian path.

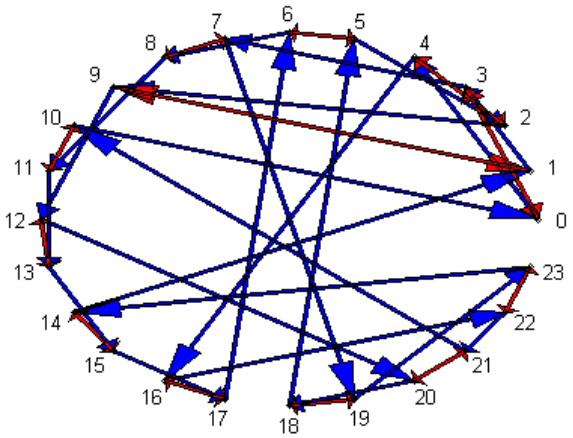
```
> HamPath({a,b},{[a,a],[b,b,b],[a,b,a,b,b,a,b,a,b,b]});
```
NO PATH TRY ANOTHER VERTEX

A generating set for Z_2 x A_4 was found where the Cayley digraph does not have a Hamiltonian path. Is there a generating set where Z_2 x A_4 does have a Hamiltonian path? This is a difficult question because there are hundreds of generating sets for Z_2 x A_4 and finding a minimal one for a large group is tough. Using Groups32 two alternate generating sets were found. They both have two elements, which implies that they are minimal. Does Z_2 x A_4 with these two generating sets have a Hamiltonian path. The representation of Z_2 x A_4 with the two new generating sets will be called CayN, and CayM. First look at CayN.

```
> CayN := [{$0..23},2,[\
        {[0,3],[1,9],[2,4],[3,0],[4,2],[5,6]\
       ,[6,5],[7,8],[8,7],[9,1],[10,11],[11,10]\
       ,[12,13],[13,12],[14,15],[15,14],[16,17]\
       ,[17,16],[18,19],[19,18],[20,21],[21,20]\
       ,[22,23],[23,22]}\
       ,{[0,4],[1,3],[2,9],[3,7],[4,16],[5,2]\
```

```
        ,[6,8],[7,19],[8,11],[9,12],[10,0]\
        ,[11,13],[12,20],[13,15],[14,1],[15,17]\
        ,[16,22],[17,6],[18,5],[19,23],[20,18]\
        ,[21,10],[22,21],[23,14]}\
                  ]] :
> Hpath(CayN);
      [8, 6, 17, 15, 13, 11, 10, 21, 20, 12, 9, 1, 14, 23, 22, 16, 4, 2, 5, 18, 19, 7, 3, 0]

>Cgraph(CCay[63]);
```

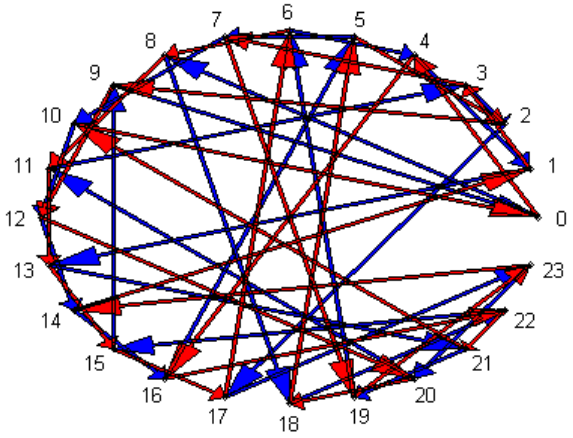


The Cayley digraph of CayN has a Hamiltonian path. Call the two elements in the generating set a and b. If a corresponds to the red arrow and b corresponds to the blue arrow then a has order 2, and b has order 6.

Next look at CayM

```
> CayM :=
[{$0..23},2,[{[0,4],[1,3],[2,9],[3,7],[4,16],[5,2],[6,8],[7,19],[
8,11],[9,12],[10,0],[11,13],[12,20],[13,15],[14,1],[15,17],
  [16,22],[17,6],[18,5],[19,23],[20,18],[21,10],[22,21],[23,14]}
,{[0,8],[1,13],[2,17],[3,2],[4,1],[5,7],[6,4],[7,10],[8,18],
  [9,0],[10,12],[11,3],[12,14],[13,21],[14,16],[15,9],[16,5],
  [17,23],[18,22],[19,6],[20,11],[21,19],[22,15],[23,20]}]] :
> Hpath(CayM);
      [10, 21, 22, 18, 20, 23, 17, 15, 13, 11, 8, 6, 19, 7, 5, 16, 14, 12, 9, 2, 3, 1, 4, 0]

> Cgraph(CayM);
```



CayM also has a Hamiltonian path. Both elements in the generating set are of order 6. This Cayley digraph is very distorted to view. As the order of the elements in the generating set gets large, there are more edges in the Cayley digraph. As the amount of edges in the Cayley digraph grows it is more difficult to see what is going on within the digraph.

Finding one generating set for $Z_2 \times A_4$ that has a Hamiltonian path and one that doesn't raises questions about what it means for a Hamiltonian path to exist in a group. Holsztynski and Strube define a group to be sequential if every generating set has a Hamiltonian path in the corresponding Cayley digraph (see [3]). Therefore $Z_2 \times A_4$ is not sequential. Holsztynski and Strube also state that every finite group of order 15 or less is sequential (see[3]). If this statement is to be extended, then one would have to look at all generating sets of all non-Abelian groups with orders between 15 and 24. Holsztynski and Strube also state that S_5 with a 2 cycle and a 5 cycle will not have a Hamiltonian path, but four two cycles might. However S_5 has order 120 so it will not extend the theorem, but it would be another useful example.

Conclusion

There are many more questions that exist when studying Hamiltonian paths in Cayley digraphs. What effects do the size of the generating set or number of edges have on the amount of time it takes to find (or not find) a Hamiltonian Path in a Cayley digraph? Does the order of the generators have anything to do with execution time? Can the theorem about all groups of order 15 or less by Holsztynski and Strube be extended, if not, what is special about groups of order 16, and which group of order 16 has a generating set that does not have a Hamiltonian path? These are lingering questions that must wait for another day. In conclusion this paper has demonstrated the interplay between programming and mathematics. The connection between the two is necessary when looking at Hamiltonian paths in the Cayley digraphs of Algebraic groups.

References

[1] Bang-Jensen, Gutin, *Digraphs: Theory, Algorithms and Applications*, Springer-Verlag London Limited (2001).

[2] Gallian J.A., *Contemporary Abstract Algebra*, 4th ed. Houghton Mifflin: New York (1998).

[3] Holsztynski, Strube, "Paths and Circuits in Finite Groups"*, Discrete Mathematics*, 22 (1978): 263-272.

[4] Maple 6, On-line Help on Groups, Waterloo Maple, Inc, 1981-2000.

[5] Maple 6, On-line Help on Networks, Waterloo Maple, Inc, 1981-2000.

[6] Maple 6, On-line Help on Plottools, Waterloo Maple, Inc, 1981-2000.

[7] Wavrik, John., Groups32 version 6.3.2a, 1990-2001, **http://math.ucsd.edu/~jwavrik**

[8] **http://www-groups.dcs.st-andrews.ac.uk/~history/Quotations/Cayley.html**

[9] **http://www-groups.dcs.st-andrews.ac.uk/~history/Quotations/Hamilton.html**

Appendix A

This appendix contains a description of all of the programs written. For each program the calling sequence is identified, the parameters are given, a description is given, the code is displayed, and there are examples for the user.

In order to use the following procedures, several of the existing packages in Maple need to be called.

Maple Packages

```
> with (networks):
> with (group):
> with(plottools):
  with(plots):
```

eelements - elements of an algebraic group.

Calling Sequence:

eelements(G);

Parameters:

G - a group, permgroup or grelgroup

Description:

- Given a group G, this routine returns the elements of the group G.
- If a permutation group is given, the elements are permutations. If a grelgroup is given then the elements are sequences of the elements in the generating set.

Code:

```
> eelements := proc(G);
   if op(0,G)=`grelgroup` then
       cosets(subgrel({},G))
   elif op(0,G)=`permgroup` then
       cosets(G,permgroup(op(1,G),{}))
   else ERROR(`argument is not a group`) fi; end:
```

Examples:

```
> eelements(permgroup(6,{[[1,2,3]],[[4,5,6]]}));
 {[ ], [[1, 2, 3]], [[4, 5, 6]], [[1, 3, 2]], [[1, 3, 2], [4, 5, 6]], [[4, 6, 5]],
      [[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 6, 5]], [[1, 3, 2], [4, 6, 5]]}

> eelements(grelgroup({a,b},{[a,a],[b,b,b],[a,b,a,b,a,b]}));
 {[ ], [b, b], [a, b], [a, b, a], [a], [b], [a, b, a, b, b], [b, a], [b, a, b], [a, b, b], [a, b, a, b],
      [a, b, a, b, b, a]}
```

DefGraph - takes a permutation group and converts it to a graph.

Calling Sequence:
DefGraph(n, gen);

Parameters:
n - the index of a permutation group
gen - a generating set of the group

Description:

- Given the arguments that are normally passed to the command permgroup, this routine returns a graph called cay1. The elements of the permutation group are the vertices of cay1 and there is an edge from a to b if as = b for some s in the generating set.

Code:

```
> DefGraph := proc(n,gen)
local s, PG, i, j:
new (cay1):
PG:= permgroup (n, gen);

s:=eelements(PG):
addvertex(s,cay1):

for i in gen do: for j in s do :
  connect ({j}, {mulperms (j,i)}, names=[[j,i]],
  'directed' , cay1)
od :od ;
cay1;
end:
```

Example:

```
> DefGraph(6,{[[1,2,3]],[[4,5,6]]});
```

cay1

MakePG - converts a grelgroup to a permutation group.

Calling Sequence:
MakePG(gens, rels);

Parameters:
gens - a set of names taken to be the generators of the group
rels - a set of relations among the generators which define the group

Description:

- Given a set of generators and a set of relations, this routine returns a permutation group.
- If the grelgroup has n elements the permutation group returned will be a subgroup of S_n .

Code:

```
> MakePG:= proc(gens,rels) local G,GR,n,ident;
  GR:= grelgroup(gens,rels);
  G:=  permrep(subgrel({},GR));
  n:=  op(1,G);
  permgroup(n,map(x->op(2,x),op(2,G)));
end:
```

Example:

```
> MakePG({a,b},{[a,a],[b,b,b],[a,b,a,b,a,b]});
 permgroup( 12, {[[1, 3, 4], [2, 5, 8], [6, 7, 11], [9, 10, 12]],
      [[1, 2], [3, 9], [4, 7], [5, 6], [8, 10], [11, 12]]} )
```

continuepath –takes a path and a graph and tries to add another element to the path.

Calling Sequence:

eelements(G, path);

Parameters:

G - a finite graph
path - a list of vertices

Description:

- Given a started path and a graph, this routine returns a list where the first element of the list is a path and the second element of the list is the word true or false.
- The path that is returned is the path that was given with another vertex added to the path if the last element of the path had a viable departure. It the last element does not have a viable departure then the path passed in is passed out without any change to the path.
- A true is returned if the path was altered and a false is returned if the path was unaltered.

Code:

```
> continuepath := proc(Graph,path)
  local daugh;
  daugh:=sort([op(departures(path[1],Graph) minus {op(path)})]);

    if daugh = []
      then RETURN ([path,false]);
      else RETURN ([[daugh[1],op(path)],true]);
    fi;
end:
```

Examples:

```
> A := graph({$1..5},{[1,2],[2,4],[1,3],[1,4],[4,1],[3,5]}):
> continuepath(A,[1]);
                                  [[2, 1], true]

> continuepath(A,[5]);
                                  [[5], false]
```

backtrack – given a path and a graph it takes one or more elements off the path

Calling Sequence:

backtrack(G, path);

Parameters:

G - a graph
path - a list of vertices

Description:

- Given a started path and a graph, this routine returns a list where the first element of the list is a path and the second element of the list is the word true or false.
- The path that is returned is the path that was passed by the user without one, two,..., or all of the last vertices. This routine continues to take vertices off the path as long as the path can not be moved forward.
- A true is returned if the path can now more forward and a false is returned if there are no elements left in the path.

Code:

```
> backtrack := proc( Graph, origpath)
  local bad, daugh, position, path ;
  path := origpath;
  do
    bad:=path [1];
    path:=path [2..nops(path)];
    if path = []
      then RETURN ([path, false]);
    fi;
    daugh:=sort([op(departures(path[1],Graph) minus                    {op(path)})]);
    member (bad,daugh,'position');
    if position < nops(daugh)
      then RETURN ([[daugh[position + 1],op(path)],true]);
    fi;
   od;
end:
```

Example:

```
> A := graph({$1..5},{[1,2],[2,4],[1,3],[1,4],[4,1],[3,5]}) :
> backtrack(A,[2,1]);
```

[[3, 1], *true*]

HamiltonianPath -determines whether a Hamiltonian path exists in a graph from a given starting point.

Calling Sequence:

HamiltonianPath(G, start);

Parameters:

G - a graph
start - a vertex

Description:

- Given a graph G and a starting vertex s, this routine returns a Hamiltonian path if one exists from the given starting point. If no path exists it prints "NO PATH TRY ANOTHER VERTEX" and returns the last path found.

Code:

```
> HamiltonianPath := proc(Graph,start)
  local path, result;
  path := [start];
  do
    do
      result := continuepath (Graph,path);
      path := result [1];
      if result [2] = false
        then break;
      fi;
    od ;

    if nops(result [1]) = nops(vertices(Graph))
      then RETURN (path);
    fi ;

    result:=backtrack (Graph,path);
    if result[2]=false
      then print (`NO PATH TRY ANOTHER VERTEX`); break ;
      else path :=result [1];
    fi;
  od;
end:
```

Examples:

```
> A := graph({$1..5},{[1,2],[2,4],[1,3],[1,4],[4,1],[3,5]}) :
> HamiltonianPath(A,1);
```

NO PATH TRY ANOTHER VERTEX

[4, 1]

```
> HamiltonianPath(A,2);
```

[5, 3, 1, 4, 2]

HPFromPerm -finds a Hamiltonian path (or no path) given a permutation group.

Calling Sequence:

HPFromPerm(n, gen);

Parameters:

n - the index of a permutation group
gen - a generating set of the group

Description:

- Given the index and a generating set of a group, this routine returns a Hamiltonian path if one exists. It no such path exists then the routine prints "NO PATH TRY ANOTHER VERTEX" and returns the last path it could find.
- The path is returned as a list of permutations, and the path always starts at the identity element.

Code:

```
> HPFromPerm:= proc(number,gen)
local Graph;
Graph:= DefGraph(number,gen);
HamiltonianPath(Graph,[]);
end:
```

Example:

```
> HPFromPerm(3,{[[1,2]],[[1,2,3]]});
        [[[1, 3, 2]], [[1, 2, 3]], [[2, 3]], [[1, 3]], [[1, 2]], [ ]]
```

HPFromRels -finds a Hamiltonian path (or no path) given a generating set and a set of relations of a group.

Calling Sequence:

HPFromRels(gens, rels);

Parameters:

gens - a set of names taken to be the generators of the group
rels - a set of relations among the generators which define the group

Description:

- Given a generating set and a set of relations for a group, this routine returns a Hamiltonian path if one exists. It no such path exists then the routine prints "NO PATH TRY ANOTHER VERTEX" and returns the last path it could find.
- The grelgroup is converted to a permutation group and the path is returned as a list of permutations. The path always starts at the identity element.

Code:

```
> HPFromRels:= proc(gens,rels) local begin, Group,Graph;
Group:= MakePG(gens,rels);
Graph:= DefGraph(op(Group));
HamiltonianPath(Graph,[]);
end:
```

Example:

```
> HPFromRels({a,b},{[a,a,a],[b,b],[a,b,a,b]});
 [[[1, 3, 2], [4, 5, 6]], [[1, 2, 3], [4, 6, 5]], [[1, 5], [2, 6], [3, 4]],
      [[1, 6], [2, 4], [3, 5]], [[1, 4], [2, 5], [3, 6]], [ ]]
```

HamPath -finds a Hamiltonian path (or no path) given a generating set and a set of relations of a group, or given the index and a generating set of a permutation group.

Calling Sequence:

HamPath(gens, rels);
HamPath(n, gen);

Parameters:

gens - a set of names taken to be the generators of the group
rels - a set of relations among the generators which define the group
n - the index of a permutation group
gen - a generating set of the group

Description:

- Given the arguments that are usually passed to permgroup or grelgroup, this routine returns a Hamiltonian path if one exists. It no such path exists then the routine prints "NO PATH TRY ANOTHER VERTEX" and returns the last path it could find.
- The path returned is a sequence of permutations.

Code:

```
> HamPath:=proc()  local r;
r:=nargs;
if type(args[1],set)
then HPFromRels(args)
else HPFromPerm(args)
fi; end:
```

Examples:

```
> HamPath({a,b},{[a,a,a],[b,b],[a,b,a,b]});
 [[[1, 3, 2], [4, 5, 6]], [[1, 2, 3], [4, 6, 5]], [[1, 5], [2, 6], [3, 4]],
      [[1, 6], [2, 4], [3, 5]], [[1, 4], [2, 5], [3, 6]], [ ]]

> HamPath(3,{[[1,2]],[[1,2,3]]});
                    [[[1, 3, 2]], [[1, 2, 3]], [[2, 3]], [[1, 3]], [[1, 2]], [ ]]
```

ngenCrep -converts a permutation group to a list where the generators are separated, so that the group can be graphed using Cgraph.

Calling Sequence:

ngenCrep(n, gen);

Parameters:

n - the index of a permutation group
gen - a generating set of the group

Description:

- Given the arguments usually passed to permgroup, this routine returns a list. The first element of the list is a set of elements of the group, the second element is the number of elements in the generating set, and the third element is another list. This sub-list is a list of sets where each set contains all the edges that correspond to one element of the generating set.

Code:

```
> AllEdges := proc(gens,elems) local gg,L;
    L := [];
    for gg in gens do
    L := [op(L),OneGen(gg,elems)]  od;
    L end:
> ngenCrep:=proc (n, gen) local PG,s,G;
 PG:=permgroup(n,gen);
 s:=eelements(PG);
 G := [s,nops(gen),AllEdges(gen,s)];
 G end:
```

Example:

```
> ngenCrep(6,{[[1,2,3]],[[4,5,6]]});
 [{[ ], [[1, 2, 3]], [[4, 5, 6]], [[1, 3, 2]], [[1, 3, 2], [4, 5, 6]], [[4, 6, 5]],
      [[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 6, 5]], [[1, 3, 2], [4, 6, 5]]}, 2, [{
      [[[1, 2, 3]], [[1, 3, 2]]], [[[1, 3, 2]], [ ]], [[[4, 5, 6]], [[1, 2, 3], [4, 5, 6]]],
      [[[1, 3, 2], [4, 5, 6]], [[4, 5, 6]]], [[ ], [[1, 2, 3]]],
      [[[4, 6, 5]], [[1, 2, 3], [4, 6, 5]]], [[[1, 2, 3], [4, 5, 6]], [[1, 3, 2], [4, 5, 6]]],
      [[[1, 2, 3], [4, 6, 5]], [[1, 3, 2], [4, 6, 5]]], [[[1, 3, 2], [4, 6, 5]], [[4, 6, 5]]]}, {
      [[[4, 6, 5]], [ ]], [[[1, 3, 2], [4, 6, 5]], [[1, 3, 2]]],
      [[[1, 2, 3], [4, 6, 5]], [[1, 2, 3]]], [[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 6, 5]]],
      [[[1, 3, 2], [4, 5, 6]], [[1, 3, 2], [4, 6, 5]]], [[ ], [[4, 5, 6]]],
      [[[1, 2, 3]], [[1, 2, 3], [4, 5, 6]]], [[[4, 5, 6]], [[4, 6, 5]]],
      [[[1, 3, 2]], [[1, 3, 2], [4, 5, 6]]]}]]
```

vertgenCrep -converts a set of vertices and a set of generators to a notation where the generators are separated, so that the group can be graphed using Cgraph.

Calling Sequence:

vertgenCrep(vert, gen);

Parameters:

vert - a set of vertices, or elements of a group.

gen - a set of generators

Description:

- Given a set of vertices and a set of generators, this routine returns a list. The first element of the list is a set of the group elements, the second element is the number of elements in the generating set, and the third is another list. This sub-list is a list of sets where each set contains all the edges that correspond to one element of the generating set.

Code:

```
> vertgenCrep:=proc(vert,gens) local M;
M:=[vert,nops(gen),AllEdges(gen,vert)];
M end:
```

Hpath -finds a Hamiltonian path (or no path)

Calling Sequence:

Hpath([vert, n, [g]]);

Parameters:

vert - a set of vertices of a graph, or elements of a group.
n - the number of elements in the generating set
[g] - a list where the n-th element is all edges corresponding to the n-th element in the generating set.

Description:

- Given the above list, this routine returns a Hamiltonian path if one exists. It no such path exists then the routine prints "NO PATH TRY ANOTHER VERTEX" and returns the last path it could find.
- The path returned is a list of elements in vert.

Code:

```
> OneGen:=proc(gg,s)local L,ss;
L:={};
for ss in s do L:=`union`(L, {op(L),[ss,mulperms(ss,gg)]});
od; L; end:
> Hpath:=proc(CayStruc) local N,P;
N:= graph(CayStruc[1],convert(map(op,CayStruc[3]),set));
P:=HamiltonianPath(N,op(1,CayStruc[1]));
P end:
```

Example:

```
> Z:=[{0,1,2,3,4,5},1,[{[0,1],[1,2],[2,3],[3,4],[4,5],[5,0]}]];
 Z := [{0, 1, 2, 3, 4, 5}, 1, [{[3, 4], [0, 1], [4, 5], [5, 0], [1, 2], [2, 3]}]]

> Hpath(Z);
                          [5, 4, 3, 2, 1, 0]
```

Cgraph -draws the Cayley digraph of a group.

Calling Sequence:

Cgraph([vert, n,[gen]]);

Parameters:

vert - a set of vertices of a graph, or elements of a group.

n - the number of elements in the generating set

[g] - a list where the n-th element is all edges corresponding to the n-th element in the generating set.

Description:

- Given the above arguments, this routine draws the Cayley digraph of a group where the edges corresponding to different generators are different colors.
- To change the colors of the graph, change the colors in the set "Colors". A list of possible colors can be found at ?plot,color.
- To change the line width change the number assigned to conc_a, to change the arrow width change the number assigned to conc_b, and to change the arrow length change the number assigned to conc_c.

Code:

conc

The areas that are in green are the code that was changed from the original concentric command.

```
> conc := proc(partitions::specfunc(list, Concentric),
            G::GRAPH, Offset::list(numeric),
            xrng::name, yrng::name)
    local n, pos, j, t, t1, orbit, y, v, x, e,
          pos1, radius, center, rotation, vset,
          lines, points, text;
    global conc_a, conc_b, conc_c, conc_clr;
    option `Copyright (c) 1992 by the University of Waterloo.`;
  radius := 0;
  center := Offset[1 .. 2];
  vset := {};
  n := 0;
  for t in partitions do
     n := n + 1;
     if not type(t, list) then
        ERROR(`partition should be a list of vertices`)
     fi;
     rotation[n] := 0;
     userinfo(3, networks, `working on`, t);
     t1 := select(has, t, 'offset');
     t :=
        select(proc(x, y) not has(x, y) end, t, 'offset')
        ;
     if t1 <> [] then
        if not type(t1, 'list'(`=`)) then
```

```
            ERROR(`usage: offset = 3.2`)
         fi;
         rotation[n] := subs({op(t1)}, 'offset');
         t := select(proc(x) not hastype(x, `=`) end, t)
      else
         if {op(t)} intersect vset <> {} then
            ERROR(`intersecting partitions involving`, t)
         fi;
         vset := vset union {op(t)}
      fi;
      if not type(t, 'list'('VERTEX'(G))) then
         ERROR(`not a list of vertices`, t)
      fi;
      orbit[n] := t
   od;
   if networks['vertices'](G) minus vset <> {} then
      n := n + 1;
      orbit[n] :=
         sort([op(networks['vertices'](G) minus vset)]);
      rotation[n] := 0
   fi;
   points := table();
   text := table();
   pos := table();
   if n = 1 and nops({op(orbit[1])}) = 1 then
      pos[orbit[1]] := center
   else for j to n do
         if 0 < radius then radius := 5/3*radius
         else radius := 1
         fi;
         pos1 := `draw/position`(orbit[j], radius, center,
            rotation[j]);
         for v in orbit[j] do
            pos[v] := pos1[v];
            points[v] := POINTS(pos[v]);
            text[v] :=
               'TEXT'(1.1*pos[v], convert(v, string))
         od
      od
   fi;
   lines := table();
   for e in edges(G) do
      x := networks['ends'](e, G);
      if 1 < nops(x) then y := x[2]; x := x[1]
      else x := x[1]
      fi;
      lines[e] := arrow(pos[x], pos[y], conc_a, conc_b,
         conc_c, color = conc_clr)
   od;
   t := map(op,
      {entries(text), entries(points), entries(lines)});
   t1 := max(op(indets(t, numeric)));
   xrng := -t1 .. t1;
   yrng := -t1 .. t1;
   RETURN(t)
```

```
end ;
```

zdraw

The text in green is the code that was modified from the draw command.

```
>   zdraw := proc()
local yrng, xrng, G, partitions, curveset, Offset, t;
option
`Copyright (c) 1992 by J. S. Devitt. All rights reserved.`;
   Offset := [0, 0, 0];
   partitions := NULL;
   for t in [args] do
      if type(t, identical('origin' = 'list')) then
         Offset := rhs(t)
      elif type(t, 'GRAPH') then G := eval(t)
      else partitions := partitions, t
      fi
   od;
   if not type(G, 'GRAPH') then ERROR(`not a graph`) fi;
   partitions := [partitions];
   if nargs = 1 and type(G(_Draw), 'procedure') then
      t := G(_Draw)(args); if t <> FAIL then RETURN(t) fi
   elif partitions = [] then
      partitions :=
         'Concentric'(sort([op(networks['vertices'](G))]))
         ;
      curveset :=
         conc(partitions, G, Offset, 'xrng', 'yrng')
   elif type(partitions[1], specfunc('list', 'Linear')) then
      if 1 < nops(partitions) then ERROR(`not implemented`)
      fi;
      curveset := `draw/Linear`(partitions[1], G, Offset,
         'xrng', 'yrng')
   elif type(partitions[1], 'specfunc'('list', Concentric))
   then
      if 1 < nops(partitions) then ERROR(`not implemented`)
      fi;
      curveset :=
         conc(partitions[1], G, Offset, 'xrng', 'yrng')
   else ERROR(`invalid args`, [args])
   fi;
   PLOT(op(curveset), AXESSTYLE(NONE))
end;



> xdraw := proc(GR,Clr)  global conc_a,conc_b,conc_c,conc_clr;
      conc_clr := Clr;
      zdraw(GR)  end :

> conc_a:=.015:
 conc_b:=.1:
 conc_c:=.1:
> Colors := [red,blue,green,orange,yellow];
```

$$Colors := [red, blue, green, orange, yellow]$$

```
> Cgraph := proc(Cay_struc) local X,Vert,Edg,numgens,i;
                global Colors;
        Vert := Cay_struc[1];
        numgens := Cay_struc[2];
        Edg := Cay_struc[3];
        for i from 1 to numgens do
                                X[i] := xdraw(graph(Vert,Edg[i]),Colors[i])
                od;
        display(convert(X,set))  end:
```

Example:

```
> CCay[12] := [{$0..7},3,[\
        {[0,1],[1,0],[2,3],[3,2],[4,5],[5,4],[6,7]\
       ,[7,6]}\
       ,{[0,2],[1,3],[2,0],[3,1],[4,6],[5,7],[6,4]\
       ,[7,5]}\
       ,{[0,4],[1,5],[2,6],[3,7],[4,0],[5,1],[6,2]\
       ,[7,3]} ]] :

> Cgraph(CCay[12]);
```