July, 1999
This paper appeared in
Bulletin of the Institute of Combinatorics and its Applications 27 (1999), 19-25.

An Algorithm for Drawing a Graph Symmetrically

Hamish Carr and William Kocay*
Computer Science Department
St. Paul's College, University of Manitoba
Winnipeg, Manitoba, CANADA, R3T 2N2
e-mail: bkocay@cs.umanitoba.ca, hcarr@cs.ubc.ca

Abstract

A technique is described that produces symmetric drawings of a graph G in the plane. The method requries that G have a non-trivial automorphism group $\text{Aut}(G)$, and that a non-trivial symmetry $g \in \text{Aut}(G)$ has been selected in some way. The technique is found to be very effective in practice. It forms a part of the Groups & Graphs** software package.

1. Symmetric Cycles

Let G be a connected undirected simple graph on n vertices $V(G)$. If $u, v \in V(G)$, then $u \to v$ means that u is adjacent to v (and so also $v \to u$). The reader is referred to [1] for other graph-theoretic terminology. Let $\text{Aut}(G)$ denote the automorphism group of G. Given a permutation $g \in \text{Aut}(G)$, it is often possible to draw G is such a way that the symmetry g is evident in the drawing. For example, there are 3 well known drawings of the Petersen graph, each illustrating a different symmetry.

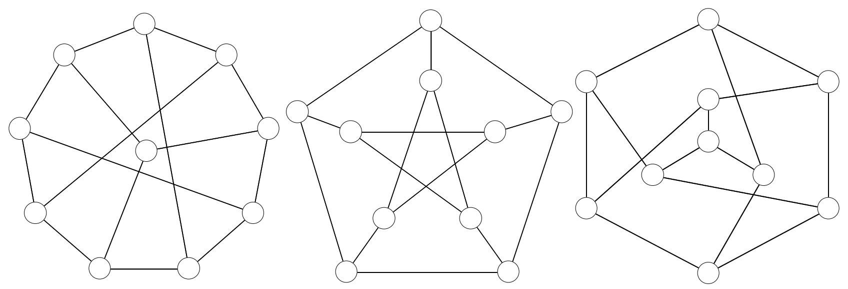


Fig. 1, 3 views of the Petersen graph

* This work was supported by an operating grant from the Natural Sciences and Engineering Research Council of Canada.

** Groups & Graphs is available on the internet, from http://bkocay.cs.umanitoba.ca/G&G/G&G.html

In this article we describe a simple algorithm that constructs a symmetric drawing of a graph, given a permutation $g \in \mathrm{Aut}(G)$.

The drawing produced will depend very much upon g. If g is a single transposition, say $g = (1, 2)$, then it will not be possible to construct a symmetric drawing without more information. Therefore we assume that g has been selected in such a way that a symmetric drawing is possible. One way to choose g is to look for permutations that contain several cycles of the same length, eg., $g = (A, B, C, D)(P, Q, R, S)(W, X, Y, Z)$. In what follows we assume that g has at least k cycles $C_1, C_2, \ldots, C_k$, each of length m, where $k \geq 1$ and $m \geq 2$. There may be other cycles as well, but we focus on these k cycles of length m. If $v \in C_i$, then we write $v + j$ for the j^{th} vertex following v in C_i. Thus the vertices of C_i can be written as $(v, v+1, v+2, \ldots, v+m-1)$, in cyclic order. The next vertex, $v+m$ would be equal to v, since calculations are performed modulo m. If $u, v \in C_i$, we write $u - v$ for the integer j such that $v + j = u$. This notation avoids the use of cumbersome subscripted subscripts. The algorithm is based on the following simple but extremely useful lemma.

1.1 Lemma. *Suppose that $C_1, C_2, \ldots, C_k$ contain vertices $v_1, v_2, \ldots, v_k$, respectively, such that $v_i \to v_{i+1}$ for $i = 1, 2, \ldots k-1$, and that $v_k \to v_1' \in C_1$, where $v_1' \neq v_1$. Then G contains a cycle C of length $km/\gcd(m, d)$, where $d = v_1' - v_1$.*

Proof. Let the vertices of C_1 be $(v_1, v_1+1, \ldots, v_1+m-1)$, in cyclic order. Let $d = v_1' - v_1$. Since $v_1 \to v_2$ and since g is a symmetry of G, we know that $v_1' \to v_2' = v_2 + d$. Similarly, $v_2' \to v_3' = v_3 + d$, etc. This sequence of adjacencies returns to C_1 every k steps. After $m/\gcd(m, d)$ such sequences of k steps, it will return to v_1. This creates a cycle C in G of length $km/\gcd(m, d)$.

In fact, if $\gcd(m, d) = i > 1$, G contains i vertex-disjoint cycles, all of the same length. Notice that g permutes these cycles in cyclic order, and that g^i rotates each cycle through k/i vertices. These cycles are termed *symmetric cycles*. If d and m are relatively prime, there is one cycle C of maximum possible length km. The first part of the algorithm is to find a symmetric cycle C of largest possible length. To this end, we utilise a recursive search.

2. The Algorithm

The first step is to construct the cyclic representation of g and to classify its cycles according to their length. Let the cycles be $C_1, C_2, \ldots, C_p$ (of all lengths). For each cycle C_i we select some $v \in C_i$ as its representative, in order to identify the cycles. v is most easily chosen as the first vertex of C_i

encountered. We keep two arrays, *theCycle*$[u]$ and *theIndex*$[u]$. As we run through the points of C_i, we set

theCycle$[u] = v$, the representative of C_i, and
theIndex$[u] = j$, where u is the j^{th} vertex of C_i.

In case $v =$ *theCycle*$[v]$, so that v is a cycle representative, we use *theIndex*$[v] =$ length of C_i, to store the length of the cycle.

With these variables we can easily tell whether two vertices u and w are in the same cycle (just test whether *theCycle*$[u] =$ *theCycle*$[w]$). If they are in the same cycle, we can determine the distance between them, as *theIndex*$[u] -$ *theIndex*$[w]$. We can also tell the length of the cycle as *theIndex* [*theCycle* $[u]$]. These arrays can be constructed in $O(n)$ steps, with one for-loop. As the algorithm progresses, the cycles will be marked either "visited" or "unvisited". This can be done by marking the representative of each cycle.

We will also use an array P to store the vertices of the current path being constructed. Here $P[i]$ is the i^{th} vertex of the path, where $i = 1, 2, \ldots$. The length of P is $\ell(P)$, the number of vertices currently on P. The main program runs through the cycles C_i. It takes the representative vertex v of C_i, marks C_i "visited", and begins building a path P from v such that all vertices of P are in different cycles of the same length. It calls a recursive procedure *ExtendPath*(v) to build P. *ExtendPath* returns *true* if it finds a cycle of maximum possible length. The representative of the cycle C_i from which the path begins is stored in a variable called the *baseCycle*. The length of this cycle is stored as *baseLength*. The path P is required to contain vertices from cycles of this length only. If there are k cycles of length m, the maximum possible cycle length is km. We store this value as *maxLength*$[m]$. We also store variables *bestCyclesSize* and *bestGCD* which contain the length of the longest cycle found so far, and the $\gcd(m, d)$ for this cycle.

The calling procedure to find a cycle follows.

```
bestCycleSize := 0   { no cycle has yet been found }
for $v$ := 1 to $n$ do begin
   if $v$ =theCycle [$v$] then begin
      { $v$ is a cycle representative }
      baseCycle := $v$
      baseLength := theIndex [$v$]
      { only consider cycles of this length }
      if baseLength > 1 then begin
         { begin a new path }
         $P[1] := v$
```

```
        $\ell(P) := 1$
        mark $v$ visited
        done := *ExtendPath* ($v$)
        { leave the cycle marked visited, so it will not be used again. }
        { Since this cycle is not allowed in future paths, }
        { decrease the maxLength for this baseLength }
        maxLength[baseLength] := maxLength[baseLength] $-$ baseLength
      end
    end
  end
  { bestCycleSize now contains the longest cycle found }
```

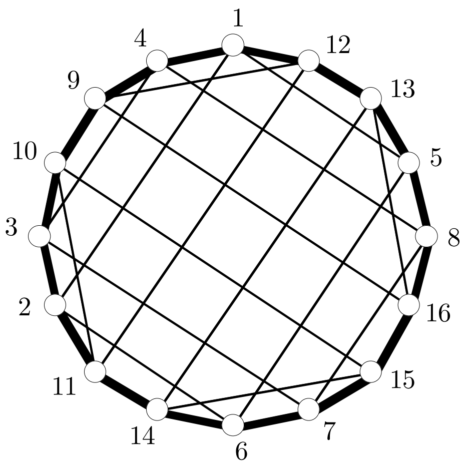
The recurscve procedure *ExtendPath* follows.

```
  **ExtendPath**($u$: vertex)
  { $P$ has length $\ell(P) \geq 1$. Extend it from vertex $u$ }
  begin
    for all $v \to u$ do begin
      $k$ := *theCycle* [$v$]
      { $v$ is in a cycle $C_k$ with representative $k$ }
      $d$ := *theIndex* [$v$]   { $d$ = length of $C_k$ }
      if $d$ = baseLength then begin
        if $C_k$ has not been visited then begin
          add $v$ to $P$, increment $\ell(P)$
          mark $v$ visited
          if ExtendPath($v$) then return(true)
          mark $v$ unvisited   { reset for next iteration }
          remove $v$ from $P$, decrement $\ell(P)$
        end
        else if $C_k$ equals baseCycle then begin
          { back to the starting cycle }
          if $v$ = *theCycle* [$v$] then begin
            { back to starting point on first return to base cycle }
            { don't accept these cycles unless there is nothing longer }
            if $\ell(P) \leq$ bestCycleSize then goto 1   { try next $v$ }
            if $\ell(P) = 2$ then goto 1   { a "cycle"of length 2 – ignore }
            save $P$ in bestPath, save $d$ in bestGCD
            save $\ell(P)$ in bestCycleSize
            goto 1   { try next $v$ }
          end
          $g$ := gcd($d$, baseLength)
          { calculate the length of the grand cycle $C$ }
          $\ell(C) := \ell(P)*$ baseLength/$g$
```

```
                { take the longest possible cycle with largest possible gcd }
                if $\ell(C) <$ bestCycleSize then goto 1   { too short − ignore }
                if $\ell(C) =$ bestCycleSize then if $g \leq$ bestGCD then goto 1
                save $P$ in bestPath, save $g$ in bestGCD
                save $\ell(C)$ in bestCycleSize
                { we may have found the longest possible cycle }
                if $\ell(C) =$ maxLength[$d$] then return(true)
            end
    1:  { try next vertex $v$ }
    end
    return(false)   { no maximum length cycle was found }
end { ExtendPath }
```

3. Sample Drawings

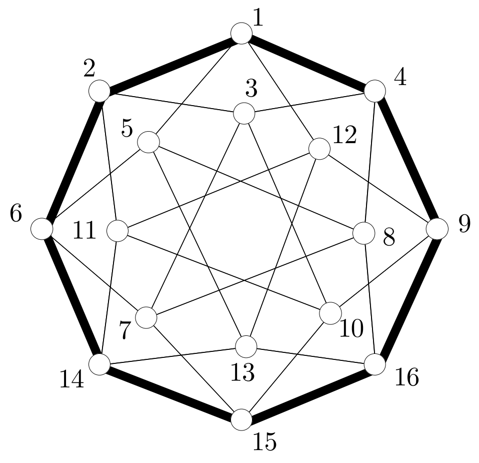
Some drawings of the 4-cube Q_4 produced by this algorithm are illustrated in Figures 2, 3 and 4. They were obtained by inputting different symmetries to the algorithm. The symmetry chosen as input is also shown. If the algorithm finds a symmetric cycle C containing all vertices of G, then the drawing produced is usually very aesthetic (see Figure 2). The vertices of the cycle are arranged in a circle. If $\gcd(m, d) > 1$ there will be more than one cycle (see Figure 3). The algorithm will then arrange the vertices of the cycles in concentric circles. In Figure 3 the vertices of the inner cycle have been manually re-arranged into a star instead of taking them in cyclic order.



(1,8,6,3)(2,4,5,7)(9,13,15,11)(10,12,16,14)

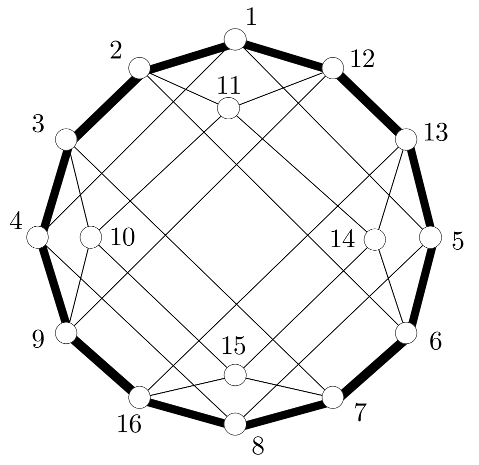
Fig. 2, View 1 of the 4-cube Q_4.

If there are some vertices not contained in the cycle(s) found, then they will be placed in the interior of the circles (see Figure 4). There are various heuristics that one could use to automatically place vertices that are not part of the cycles found. In practice it is usually easy to adjust these manually so long as there are few of them.



(1,12,9,10,15,7,6,5)(2,13,4,11,16,3,14,8)

Fig. 3, View 2 of Q_4.



(1,5,8,4)(2,13,7,9)(3,12,6,16)(10,11,14,15)

Fig. 4, View 3 of Q_4.

As currently programmed, the algorithm requires that a symmetry be input. This kind of control over the algorithm is necessary if one wants to construct drawings highlighting a certain symmetry. If one wants to automate the process further, one way to do it would be to use an algorithm which finds the conjugacy classes of $\mathrm{Aut}(G)$, and then selects a permutation from one of the conjugacy classes, so that a particular cycle structure is chosen, eg, the largest possible number of vertices lie in cycles of the same length. Permutations of this type are often found to produce very good drawings in practice. However conjugacy classes can be difficult to compute. A method that works well in practice is to select a number of random permutations from $\mathrm{Aut}(G)$ and count the number of points in cycles of the same length. Select a permutation that gives the largest possible such count, with the largest possible cycle length. Many variations are possible.

References

1. J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications*, American Elsevier Publishing, New York, 1976.
2. William Kocay, "Groups & Graphs, a Macintosh application for graph theory", Journal of Combinatorial Mathematics and Combinatorial Computing 3 (1988), 195-206.