A note on `Is2EdgeTransitive`

Frankie Gillis

October 2025

1 Introduction

Recall that a digraph is an ordered pair $\Gamma = (V, E)$, where V is the vertex set, and $E \subseteq V \times V$ is the edge set, which is a collection of ordered pairs of elements of V. The automorphism group $\mathrm{Aut}(\Gamma)$ of Γ is the group of permutations of V which fix E setwise.

Definition 1.1. Let Γ be a digraph. A 2-*edge* in Γ is a triple (u, v, w) of distinct vertices such that both $(u, v) \in E$ and $(v, w) \in E$. The set of all 2-edges in Γ is denoted $T(\Gamma)$.

An example of a 2-edge is given in Figure 1, and examples of non-2-edges are given in Figure 2.

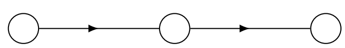

Figure 1: A 2-edge.

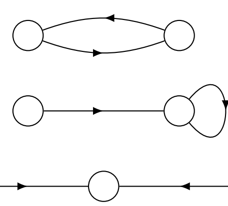

Figure 2: Not 2-edges.

Definition 1.2. Let Γ be a digraph. Then Γ is called 2-*edge transitive* if the induced action of $\mathrm{Aut}(\Gamma)$ on $T(\Gamma)$ is transitive. That is, given any pair of 2-edges (x, y, z) and (u, v, v) in Γ, there exists an automorphism φ of Γ such that

$$(\varphi(x), \varphi(y), \varphi(z)) = (u, v, w).$$

Definition 1.3. Let Γ be a digraph. A 2-*cycle* in Γ is a triple (u, v, u) of vertices such that $(u, v), (v, u) \in \Gamma$. We say that (u, v, u) is a 2-cycle *based at* v.

2 Determining 2-edge transitivity

Determining 2-edge transitivity of a digraph Γ takes place in two steps: enumerate all 2-edges in Γ, then calculate the orbit length of a single 2-edge under $\mathrm{Aut}(\Gamma)$. Then Γ is 2-edge transitive if and only if these numbers are equal. The orbit calculation is the most computationally-intensive step in the algorithm, and in general its time complexity is difficult to estimate. In practice, we calculate the stabiliser of a 2-edge, and then utilise the Orbit-Stabiliser theorem and Cayley's theorem to find the orbit length.

Thus, we are left with the task of enumerating the 2-edges of Γ as efficiently as possible.

2.1 Enumerating 2-edges

Let Γ be a digraph with $|V(\Gamma)| = n$ and $|E(\Gamma)| = m$. Naively, the operation of computing the number of 2-edges in Γ has complexity $\mathcal{O}(n^3)$: we iterate over all n^3 elements of $V \times V \times V$ and increment the count every time an element satisfies the conditions to be a 2-edge. **GAP**-style pseudocode for this is given in Algorithm 1.

Algorithm 1 Naive implementation

```
procedure Is2EdgeTransitive(D)
    O := OutNeighbours(D);
    n := Length(O);
    twoEdges := [];
    for u in [1 .. n] do
        for v in O[u] do
            for w in O[v] do
                if u <> v and v <> w and w <> u then
                    Add(twoEdges, [u, v, w]);
                end if
            end for
        end for
    end for
    numTwoEdges := Length(twoEdges);
    if numTwoEdges = 0 then
        return true;
    else
        G := AutomorphismGroup(D);
        return Length(twoEdges) * Order(Stabiliser(G, twoEdges[1])) = Order(G);
    end if
end procedure
```

In this section, we introduce a few definitions and lemmas which we will later use to significantly improve the performance of Algorithm 1.

Definition 2.1. Let $t = (u, v, w)$ be a 2-edge in a digraph Γ. The vertex v is called the *center* of t. The set of all 2-edge centers in Γ is denoted $C(\Gamma)$.

Lemma 2.2. *If Γ is 2-edge transitive, then* $\mathrm{Aut}(\Gamma)$ *acts transitively on* $C(\Gamma)$.

Proof. Let v, $v' \in C(\Gamma)$. Then, there exist 2-edges (u, v, w), (u', v', w') in Γ for some vertices u, u', w, $w' \in V(\Gamma)$. Since Γ is 2-edge transitive, there is an automorphism φ of Γ such that

$$(\varphi(u), \varphi(v), \varphi(w)) = (u', v', w').$$

In particular, $\varphi(v) = v'$. □

Unfortunately, the converse of Lemma 2.2 is not true. Counterexamples to this are provided in Example 2.3 and Example 2.4.

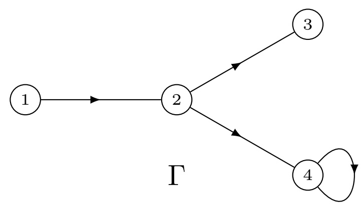

Figure 3: The digraph Γ.

Example 2.3. By inspection, we can see that $T(\Gamma) = \{(1, 2, 3), (1, 2, 4)\}$. Therefore $C(\Gamma) = \{2\}$ and so $\mathrm{Aut}(\Gamma)$ clearly acts transitively on $C(\Gamma)$ via the identity automorphism. However, any automorphism mapping the 2-edge $(1, 2, 3)$ onto the 2-edge $(1, 2, 4)$ must map 3 onto 4, which is impossible as 3 has no loop while 4 has a loop. It follows that Γ is not 2-edge transitive.

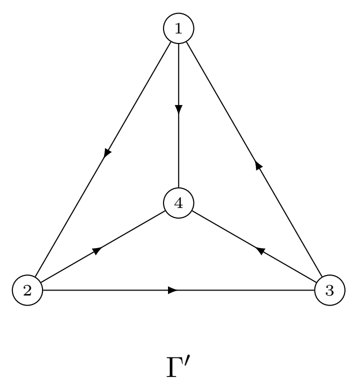

Figure 4: The digraph Γ'.

Example 2.4. First, note that $\mathrm{Aut}(\Gamma') = \langle (1\,2\,3) \rangle$, the cyclic group of order 3 generated by the permutation $(1\,2\,3)$. By inspection, $C(\Gamma') = \{1, 2, 3\}$, and so $\mathrm{Aut}(\Gamma')$ acts transitively on $C(\Gamma')$. However, note that there can be no automorphism of Γ' which maps the 2-edge $(1, 2, 3)$ onto the 2-edge $(1, 2, 4)$, as such a permutation must map the vertex 3 onto the vertex 4 which is impossible as $\deg_{\mathrm{out}}(3) \neq \deg_{\mathrm{out}}(4)$. Hence, Γ' is not 2-edge transitive.

If the converse of Lemma 2.2 were true, we could compute 2-edge transitivity by determining whether $\mathrm{Aut}(\Gamma)$ acts transitively on $C(\Gamma)$. This is advantageous as determining the latter has the complexity of computing vertex transitivity, which is much quicker than 2-edge transitivity. We do, however, get the following corollary which will become useful.

Corollary 2.5. *If Γ is 2-edge transitive, then for any u, $v \in C(\Gamma)$:*

1. $\deg_{in}(u) = \deg_{in}(v)$*; and*
2. $\deg_{out}(u) = \deg_{out}(v)$*.*

Proof. Follows from the fact that any automorphism of a digraph must preserve in-degree and out-degree. ☐

Proposition 2.6. *Let Γ be a 2-edge transitive digraph. Then, there exist positive integers p, q and non-negative integers l, c such that for every $u \in C(\Gamma)$*

1. $\deg_{in}(u) = p$*;*
2. $\deg_{out}(u) = q$*;*
3. *There are l loops at u; and*
4. *There are c 2-cycles at u.*

Proof. Let $u \in C(\Gamma)$ be any 2-edge center. Let $p = \deg_{\text{in}}(u)$, $q = \deg_{\text{out}}(u)$, let l be the number of loops at u and c be the number of 2-cycles at u.

Let $v \in C(\Gamma)$. By Corollary 2.5, $\deg_{in}(v) = p$ and $\deg_{\text{out}}(v) = q$. Let φ be an automorphism of Γ such that $\varphi(u) = v$. Such an automorphism exists since $\text{Aut}(\Gamma)$ acts transitively on $C(\Gamma)$. The loop $(u, u) \in E(\Gamma)$ if and only if $(\varphi(u), \varphi(u)) = (v, v) \in E(\Gamma)$, so there are l loops based at v.

Similarly, for any $w \in V(\Gamma)$, the 2-cycle (w, u, w) exists in Γ if and only if $(w, u), (u, w) \in E(\Gamma)$. This happens if and only if $(\varphi(w), v), (v, \varphi(w)) \in E(\Gamma)$, so the 2-cycle (w, u, w) is in Γ if and only if $(\varphi(w), v, \varphi(w))$ is in Γ. Since there are c such 2-cycles at u, it follows that there are c 2-cycles at v.

☐

Proposition 2.7. *Let Γ be a 2-edge transitive digraph. Then,*

$$|T(\Gamma)| = [(p - l)(q - l) - c] \, |C(\Gamma)|.$$

Proof. First, we count the number of 2-edges in $T(\Gamma)$ whose center is v. Since $\deg_{\text{in}}(v) = p$, there are $p - l$ edges leading into v which are not loops. Similarly, there are $q - l$ edges leading out of v which are not loops. This gives $(p - l)(q - l)$ ways to choose an ordered triple (u, v, w) where neither (u, v) nor (v, w) are loops.

Now we must account for 2-cycles of the form (u, v, u) which have been counted. By assumption, there are c such 2-cycles, and these must be subtracted from the total. Therefore, there are

$$(p - l)(q - l) - c$$

2-edges with v as their center.

By the previous proposition, this is true for any $u \in C(\Gamma)$. It follows that

$$|T(\Gamma)| = [(p-l)(q-l)-c]\,|C(\Gamma)|.$$

$\square$

2.2 Determining l

Let l be the number of loops based at any $u \in C(\Gamma)$. Since we are not considering multidigraphs, $l \in \{0,1\}$, as either $(u,u) \in E(\Gamma)$ or $(u,u) \notin E(\Gamma)$. Therefore, it suffices to check if $(u,u) \in E(\Gamma)$. This can be done in $\mathcal{O}(n)$ time by checking if u is an out-neighbour of u.

2.3 Determining c

Let c be the number of 2-cycles based at any $u \in C(\Gamma)$. That is,

$$c = \#\{(u,v,u) \mid (u,v),(v,u) \in E(\Gamma)\}$$

for any $u \in C(\Gamma)$. Equivalently,

$$c = \#(\{u \in V(\Gamma) \mid (u,v) \in E(\Gamma)\} \cap \{u \in V(\Gamma) \mid (v,u) \in E(\Gamma)\}).$$

This latter presentation is the most useful, as it says c is the size of the intersection of the in-neighbours of u and the out-neighbours of u. In **GAP**, the in-neighbours and the out-neighbours of a vertex are both ordered lists of integers, and we can compute the intersection size of two ordered lists of integers in $\mathcal{O}(n)$ time.

2.4 Determining `centers`

Let $v \in V(\Gamma)$. Set $p_0 = \deg_{\text{in}}(v)$, $q_0 = \deg_{\text{out}}(v)$, l_0 as the number of loops at v, and c_0 as the number of 2-cycles at v. By Proposition 2.7, the number of 2-edges whose center is v is

$$(p_0 - l_0)(q_0 - l_0) - c_0.$$

In particular, $v \in C(\Gamma)$ if and only if

$$(p_0 - l_0)(q_0 - l_0) - c_0 > 0.$$

3 The Algorithm

3.1 Analysis of Time Complexity

Enumeration of 2-edges in Algorithm 2 has time complexity $\mathcal{O}(n^2+m)$. The m term comes from building the InNeighbours list, as digraphs in **GAP** are stored by OutNeighbours. The n^2 term comes from looping through each vertex, which is $\mathcal{O}(n)$ time, and within each loop calculating l and c, each of which take $\mathcal{O}(n)$ time, giving $\mathcal{O}(n^2)$ time in total. This certainly beats the $\mathcal{O}(n^3)$ of Algorithm 1, since $m \leq n^2$.

Algorithm 2 Faster implementation

```
procedure Is2EdgeTransitive(D)
    O := OutNeighbours(D);
    I := InNeighbours(D);
    n := Length(O);
    p := 0;
    q := 0;
    l := 0;
    c := 0;
    centers := [];
    for u in [1 .. n] do
        p_0 := Length(I[u]);
        q_0 := Length(O[u]);
        if u in O[u] then
            l_0 := 1;
        else
            l_0 := 0;
        end if
        c_0 := IntersectionSize(I[u], O[u]);
        if (p_0 − l_0) ∗ (q_0 − l_0) − c_0 > 0 then
            Add(centers, u);
            if p = 0 then
                p := p_0;
                q := q_0;
                l := l_0;
                c := c_0;
            else
                if p <> p_0 or q <> q_0 or l <> l_0 or c <> c_0 then
                    return false;
                end if
            end if
        end if
    end for
    numTwoEdges := ((p − l) ∗ (q − l) − c) ∗ Length(centers);
    if numTwoEdges = 0 then
        return true;
    else
        v := centers[1];
        for u in I[v] do
            for w in O[v] do
                if u <> v and v <> w and w <> u then
                    twoEdge := [u, v, w];
                    G := AutomorphismGroup(D);
                    return numTwoEdges ∗ Order(Stabiliser(G, twoEdge)) = Order(G);
                end if
            end for
        end for
    end if
end procedure
```

4 Plots

4.1 Algorithm 1 versus 2