A Polynomial Time Algorithm for the Detection of Axial Symmetry in Directed Acyclic Graphs [⋆]

S. Bhowmick and P. Hovland

Mathematics and Computer Science Division, Argonne National Laboratory,
9700 South Cass Avenue, Argonne, IL 60439-4844.
E-mail {bhowmick, hovland}@mcs.anl.gov

Abstract. Computation of Hessians in automatic differentiation can be done by applying elimination techniques on a symmetric computational graph. Detection of symmetry in the graph, i.e. matching the vertices and edges with their corresponding pairs can enable us to identify duplicate mirror operations. We can exploit symmetry by performing only one of each of these duplicate operations and thus greatly reduce the computation costs, by almost halving the number of operations. In this paper we present a polynomial time algorithm that can detect symmetry in directed acyclic graphs. We prove the correctness of the algorithm and demonstrate with an example how detection of symmetry lowers the cost of computing Hessians.

1 Introduction

The Hessian of a function $f(x)$ can be calculated from a symmetric directed acyclic graph (DAG). This symmetric computational graph corresponds to the gradient computation and can be generated from the function DAG using the non-incremental reverse mode of automatic differentiation. Details of creating such symmetric graphs can be found in [2]. It has also been conjectured that maintaining symmetry of the graph while computing the Hessian might lead to near optimal number of operations. The symmetric structure of the graph can be exploited by storing only half of the graph and by not recomputing mirror operations.

In order to obtain the benefits of working with a symmetric graph, it is important to determine the axis of symmetry. It has been proven that testing symmetry of a general graph is a NP-complete problem [1, 5]. However when considering computational graphs arising in automatic differentiation, it is possible to develop a polynomial time algorithm. This is because we are working with a directed acyclic graph which provides more information than an undirected one. Furthermore we are looking for an axis of symmetry perpendicular to the direction of flow. Additional information regarding possible vertex and edge pairs can be obtained if the vertices can be demarcated as the ones corresponding with the variables in calculating the function and the ones generated during gradient calculation.

[⋆] This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy under Contract W-31-109-Eng-38.

The rest of the paper is arranged as follows. In Section 2 we define the terms and mathematical expressions that will be used subsequently. In Section 3 we describe the algorithm for detecting symmetry. Section 4 provides the analysis of the algorithm and investigates its correctness and runtime complexity. Section 5 deals with an interesting class of DAGs which have more than one line of symmetry. Section 6 provides an example from an optimization problem which shows that exploiting symmetry does indeed lower the computational costs. We conclude by discussing our future research plans regarding symmetry and Hessian calculations.

2 Preliminary Definitions

In this section we define some terms used in graph theory. Unless mentioned otherwise, the terms used here are as they are defined in [4].

A graph $G = (V, E)$ is defined as a set of vertices V and a set of edges E. An edge $e \in E$ is associated with two vertices u, v which are called its *endpoints*. If a vertex v is an endpoint of an edge e, then e is incident on v. A vertex u is a *neighbor* of v if they are joined by an edge. The set of neighbors of v is given by $Neigh(v)$.

A *directed edge* is an edge $e = (uv)$, one of whose endpoints, u is designated as the *tail* and whose other endpoint, v is designated as *head*. A directed edge is directed from its tail to its head. If (u, v) is a directed edge, then u is the *predecessor* of v and v is the *successor* of u. A *directed graph* (digraph) is a graph with directed edges.The *indegree* of a vertex v in a digraph is the number of edges directed towards v and is equal to the set of predecessors, $Pred(v)$. The *outdegree* of a vertex v in a digraph is the number of edges directed from v and is equal to the set of successors, $Suc(v)$.

A *walk* in a graph G is an alternating sequence of vertices and edges, $W = v_0, e_1 \ldots,$ e_n, v_n, such that for $j = 1, \ldots, n$, the vertices v_{j-1} and v_j are the endpoints of e_j. A walk is closed if the initial vertex is also the final vertex. A $trail$ is a walk such that no edge occurs more than once. A $path$ is a trail where no internal vertex is repeated. A closed path is called a *cycle*. A *directed acyclic graph* (DAG), is a directed graph with no cycles.

An *automorphism* [1] of an undirected graph $G = (V, E)$ is a permutation σ of V and E, such that,

1. $\sigma(E) = E$
2. $\sigma(V) = V$
3. $e \in E$ is incident to $v \in V \iff \sigma(e)$ is incident to $\sigma(v)$

The graph G is said to have *axial symmetry* if there exist an automorphism $\sigma \in Aut(G)$, such that the subgraph of G induced by the fixed point set of σ is embeddable on a line [1]

To define the *axial symmetry* in a DAG, we modify the third property of automorphism as follows;

– If the head of $e \in E$ is $v \in V$ and the tail is $u \in V$, then $\sigma(u)$ is the head and $\sigma(v)$ is the tail of $\sigma(e)$

It should also be noted that in an axial symmetric DAG, all fixed points of $\sigma \in Aut(G)$ will be included in E. The vertices in V can be divided into two groups $V1$ and $V2$, such that for all $v \in V1$ there exists $\sigma(v) \in V2$. We call such a set $v, \sigma(v)$ as a $vertex - pair$. Similarly the edges, except the fixed points,can be divided into two groups $E1$ and $E2$, such that for $v, u \in E1$ there exists $\sigma(v)\sigma(u) \in E2$. We call such a set of edges as an $edge - pair$. There may exist more than one permutation satisfying the properties for axial symmetry. The sets $V1_\sigma$ and $V2_\sigma$ are determined based on a particular permutation σ.

3 Symmetry Detection Algorithm

In this section we will describe an algorithm to determine the vertex-pair and edge-pair sets in a DAG for a particular permutation σ. We observe that the relation σ is commutative, i.e. $u = \sigma(v)$ and $v = \sigma(u)$, therefore for a particular permutation σ, vertex and edges pairs are uniquely determined. The conditions for two vertices v and u to be pairs of each other are;

1. $indegree(v) = outdegree(\sigma(v))$ and $outdegree(v) = indegree(\sigma(v))$
2. $\forall w \in Neigh(v),\ \ \sigma(w) \in Neigh(u)$

Our algorithm is based on finding vertex pairs that satisfy the above conditions.

```
Symmetry Detection Algorithm
C:All groups have equal number of real and dual vertices or even number of neutral
   vertices
STEP 1: Divide the vertices into groups according to the first condition
    Create groups $\Psi_{ij}, (i \leq j)$
    If $i = j$ then $\Psi_{ij}$ contains an array $neutral$
    If $i \neq j$ then $\Psi_{ij}$ contains arrays $real$ and $dual$
    For all vertex $v$
        $i = indegree(v)$ and $j = outdegree(v)$
        If $i < j$ then $v$ is added to $\Psi_{ij}.real$
        If $i > j$ then $v$ is added to $\Psi_{ij}.dual$
        If $i = j$ then $v$ is added to $\Psi_{ij}.neutral$
    if(C is TRUE for all $\Psi_{ij}$)
        Continue to STEP 2
    else
        Graph is nonsymmetric; break
STEP 2: Subdivide the groups according to the second condition
    Set number of groups to $n$
    while ($n > 0$)
        Sort groups $\Psi_0$ to $\Psi_n$ in increasing order of group-size
        If $\Psi_0$ contains real and dual arrays
            Set V1=$\Psi_0.real[0]$
            Set V2=$\Psi_0.dual[0]$
```

```
else
    Set V1=$\Psi_0.neutral[0]$
    Set V2=$\Psi_0.neutral[1]$
Create new groups $\bar{\Psi}_a$ corresponding to older groups $\Psi_a$ as follows;
For all neighbors $n$ of $V1$
    Remove $n$ from its original group $\Psi_a$
    Add $n$ to $real$ array in $\bar{\Psi}_a$
For all neighbors $n$ of $V2$
    Remove $n$ from its original group $\Psi_a$
    Add $n$ to $dual$ array in $\bar{\Psi}_a$
if(C is TRUE for all $\bar{\Psi}_a$)
    Add new groups $\bar{\Psi}_a$ into set of existing groups
    Remove groups with zero elements
    Set n = number of groups
else
    Graph is nonsymmetric; break
```

4 Algorithm Analysis

In this section we briefly describe how the algorithm works and offer a proof of correctness. We then demonstrate that for most graphs, the complexity of the algorithm is $O(V^2)$.

4.1 Brief Description of Algorithm

In step 1, the vertices are divided into groups according to their degrees. By the first necessary condition, it is easy to see that if vertex v is an element of group Ψ, its pair $\sigma(v)$ is also an element of Ψ. If the graph is symmetric each group Ψ consists of either equal number of real and dual vertices, stored as arrays of $\Psi.real$ and $\Psi.dual$ or an even number of neutral vertices, $\Psi.neutral$. If this condition is not satisfied then the graph is nonsymmetric, and we need not proceed further.

At the end of step 1 the vertices are divided into groups. However there may be multiple vertex pairs in each group. In step 2 we further divide the groups based on the second condition. At each iteration two vertices $V1$ and $V2$ are removed from the smallest group. If $V1$ and $V2$ form a vertex-pair, then each group Ψ_a contains a neighbor of $V1$ or $V2$, is subdivided into, i) the original Ψ_a, contains vertices not neighbors of $V1$ and $V2$ and ii) $\bar{\Psi}_a$, containing vertices which are neighbors of $V1$ or $V2$. Without loss of generality we label neighbors of $V1$ as real and neighbors of $V2$ as dual. If condition C is satisfied at the end of the iteration, we continue. Otherwise the graph is nonsymmetric. The graph is symmetric if all vertices can be paired into $V1$ or $V2$ i.e. C is satisfied for all iterations. If the graph is symmetric, then step 2 converges in at most $V/2$ iterations.

4.2 Correctness of the Algorithm

Theorem 1. *The graph G is symmetric if and only if condition C is satisfied at Step 1 and for all iterations of Step 2*

Proof. If C is satisfied at the end of Step 1, the vertices have been divided into potential vertex-pairs. We note that if a group contains only two elements, then the elements are vertex-pairs. At each iteration we select two vertices $V1$ and $V2$. Since $V1$ and $V2$ are in the same group, they satisfy the first condition. If after subdivision the new set of groups satisfy condition C, then each of their neighbors satisfy the first condition. If C is satisfied for all subsequent iterations, repeated subdivision of the groups will ultimately result in groups with only two elements. Since the groups have been formed by satisfying the first condition clearly these two elements are vertex-pairs. If for two vertices all their neighbors end up being mutually vertex-pairs, then the second condition is satisfied and $V1$ and $V2$ are vertex pairs. Thus if C is satisfied at all iterations, then all vertices can be paired.

If the graph is symmetric then C must be satisfied at end of Step 1. If the number of elements in Ψ_0 is two then $V1$ and $V2$ selected at the beginning of each iteration are vertex-pairs. If the number of elements is higher then it has been observed that an arbitrary selection of $V1$ and $V2$ from the appropriate sets satisfy the vertex-pair conditions. Therefore, their neighbors would satisfy the necessary conditions and subdivide accordingly into equal number of real and dual pairs and C would be satisfied at each iteration.

4.3 Algorithm Complexity

Step 1 takes time proportional to the number of vertices. Checking condition C takes time proportional to the total number of elements in all the groups, i.e. the number of vertices. Each iteration of Step 2 requires,

1. Sorting the groups; time is $O(n \log n)$
2. Subdividing the neighbors; time is proportional to the degree of any vertex in the vertex-pair
3. Adding new groups; time is constant
4. Checking condition C; time is $O(V)$

At the end of Step 2 all edges are traversed and the step converges in at most $V/2 = O(V)$ iterations. Therefore the running time of the algorithm is $O(V) + O(Vn \log n) + O(E) + O(V^2)$. For most graphs n is much smaller than V. Therefore the running time can be concisely expressed as $O(V^2)$.

5 Multiple Symmetric Permutations

There may exist DAGs for which symmetry can be obtained by more than one permutation. Figure 1 shows a graph where symmetry is obtained from multiple permutations. When the algorithm is applied on graphs with this property, at least in one iteration the

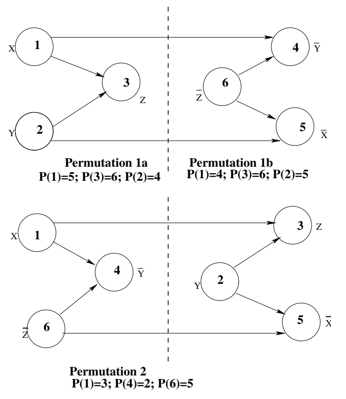

Fig. 1. Graph exhibiting more than one line of symmetry

initial group Ψ_0 has more than two elements. An arbitrary selection of $V1$ and $V2$ from the appropriate sets give the vertex-pairs corresponding to one of the permutations.

While we can obtain one line of symmetry by using the algorithm, in certain cases, for example when determining the symmetry of a computational graph, this might not be the one required. An exhaustive search for all symmetric permutations can be up to the order of $V!$. A better strategy is for the user to add extra information regarding the vertices and edges in the graph. For example the graph in Figure 1 is the computational graph for the following function and its gradient calculated in reverse mode;

- $Z = X * Y$
- $\bar{X} = \bar{Z} * Y$
- $\bar{Y} = \bar{Z} * X$

Permutation $1a$ pairs the variables X, Y and Z with their corresponding reverse $\bar{X}, \bar{Y}$ and $\bar{Z}$. Permutation $1b$ pairs X with $\bar{Y}$, Y with $\bar{X}$ and Z with $\bar{Z}$. Since both the operations are multiplications, this pairing can be utilized to exploit the symmetric calculation of the Hessian. However, if this was not the case, and the pairing of the variables with their corresponding reverses was necessary, then the vertex information should also include the type of the operation executed at that node. An even more convoluted pairing is given in permutation 2 where X is paired with Z. Such pairing of inputs with outputs can be avoided, if the vertices are differentiated by the ones used in the calculation of the function and the ones used in calculation of the gradient. In short the algorithm finds only one line of symmetry, if the user requires the vertex-pairs to follow some additional properties then this information should be included as a necessary condition. The division of vertices in step 1 will incorporate all the necessary conditions with more stringent definitions of reals and duals. The subdivision in step 2 will remain unchanged. Since step 2 is the most expensive part of the process, adding extra information does not significantly increase the running time of the algorithm.

6 Application of Symmetry Detection

We have implemented the symmetry detection algorithm within the OpenAD [3] framework, and have successfully detected symmetry for several test graphs, including ones with multiple lines of symmetry.

We have also applied the algorithm on an optimization code that checks the quality of a mesh based on the inverse-mean ratio shape quality metric [6]. We considered a two-dimensional meshing code that compares the elements with an equilateral triangle. The original hand-coded Hessian required 138 flops. The gradient evaluation needed 46 and the function required 26 flops. We created a symmetric computational graph, based on the objective function and its gradient. The symmetry detection code was successful in determining the vertex-pairs. A cross country vertex elimination based on the computational graph required an additional 54 multiplications and 10 additions. Exploiting symmetry in the calculation of the Hessian reduced the cost to 52 additional multiplications and 2 additions, for a total number of operations of 118. This is an improvement of 10 flops over non-symmetric crosscountry elimination and 20 flops over the hand-coded Hessian. The time taken to build the graph is 0.42380 seconds and time to detect symmetry is .02512 seconds, for a total of .44892 seconds.

The current implementation of the algorithm is still at its preliminary stage and several code optimizations can be added to lower the execution time. For example, we plan use the graph generated during the calculation of the gradient and not build it again for symmetry detection. It should also be observed that the time depends not only on the number of vertices and edges but also how they are connected.

7 Conclusions and Future Work

Detection of mirror operations in a symmetric graph can indeed lower the number of operations required for Hessian computation. Currently our code is only able to detect vertex-pairs. In order to exploit symmetry of operations we also need to store the operations associated with each vertex-pair. We will be extending our code to incorporate this property. There are also possibilities of modification to the algorithm. Instead of subdividing the neighbors of the current vertex pair, we can be more specific and put the predecessors and successors in two groups. This will increase the number of groups and make the convergence faster. Another modification is to add preconditions to step 1, enabling the user to add extra conditions that are specific to his needs. Incorporating the symmetry detection mechanism in AD tools will lower the computational costs. Even for a moderately large function the decrease of time in Hessian computation will be much larger than the extra time required for detecting symmetry

References

1. Hubert De Fraysseix. An heuristic for graph symmetry detection. *Lecture Notes in Computer Science, Proceedings of the 7th International Symposium on Graph Drawing*, 1731:276–285, 1999.

2. A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2000.
3. J.Utke. OpenAD: Algrithm implemnetation user guide. Technical Report ANL/MCS–TM–274, Argonne National Laboratory, IL, 2004.
4. J. L.Gross and J. Yellen. *Handbook of Graph Theory and Applications*. CRC Press, 2004.
5. J. Manning. Geometric symmetry in graphs, 1990. Ph.D. Thesis, Purdue University, New York.
6. T. S. Munson. Mesh shape-quality optimization using the inverse mean-ratio metric. Preprint ANL/MCS-P1136-0304, Argonne National Laboratory, Argonne, Illinois, 2004.

The submitted manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory (”Argonne”) under Contract No. W-31-109-ENG-38 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.