

Third Author  
email3

## 2020-06-07 22:12. Page 1 of 1-6.

The notation  $\partial^+(p)$  is used to indicate the start node of path  $p$ , and similarly  $\partial^-(p)$  is the end node of path  $p$ . [Same for this notation. I see  $\partial$  being useful, but not  $\partial^-$  -DG]  $\partial(p)$  maps to the tuple (start node of  $p$ , end node of  $p$ ).

A pair of paths  $p$  and  $q$  is called *parallel* iff their terminal nodes are the same, i.e.,  $\partial(p) = \partial(q)$ . Let  $R_{all}$  be the set of all parallel pairs of paths in the diagram.

The diagram commutes iff  $f(p)=f(q) \forall (p, q) \in R_{all}$ ; that is, the composition of maps along any path connecting  $u$  to  $v$  is independent of path choice. Here, equality of functions is defined by some oracle whose specifics will vary by target functions and domain.

### 3 BASELINE ALGORITHMS

To examine the efficacy of our proposed algorithm, we compare it to some potential alternatives. Among these are a naïve

#### 3.1 Naïve algorithm for online addition

We solve the question of online addition, where the original diagram is path independent and a new edge is added. Any implementation can verify only parallel pairs that have been affected by the addition of the new edge. Pairs that do not involve the new edge must continue to be path independent because they already belonged to the old diagram.

The cycle checking steps can also be improved upon. In the case where the identity is assumed, only elementary cycles passing through the new edge need to be verified to be equal to the identity. When the identity is not assumed, cycles that include one of the nodes of the new edge must be checked for the new exit or entry point that the new edge becomes (if the cycle includes the source of the new edge then the cycle followed by the new edge must be equal to the new edge alone and if the cycle include the sink then the new edge followed by the cycle must be equal to the new edge). Additionally, all cycles that include the new edge would need full verification like in the batch algorithm- the cycle would have to be equal to itself composed with itself, and would have to be verified for all its entry and exit points.

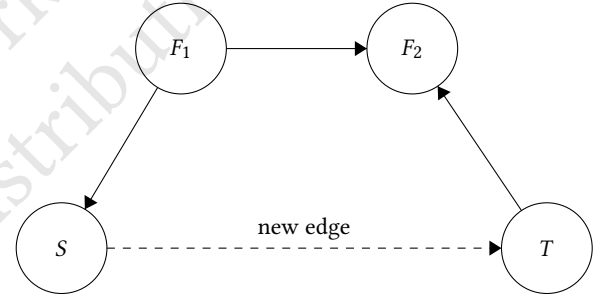
Now, the commutativity of the original diagram could be used to narrow down the set of pairs to verify. Pairs where both paths do not involve the new edge would remain equal, but also, pairs where both paths involved the new edge would have to be equal. To see why this is true, each path could be thought of as consisting of the composition of three segments, the first segment from the common source to the source of the new edge, the second, the new edge itself, and the third, from the sink of the new edge to the common sink of the parallel pair. The first segment of both pairs would have to be equal because they existed in as parallel pairs in the original diagram, and similarly the third segments would also have to be equal. The second segments, consisting of the same edge, would also have to be equal. Therefore the composition of these three equal components would be equal. Note that the new edge could only appear once because cycles have already been dealt with and none of the paths under consideration hit a node twice. Thus we are left only to verify the pairs where exactly one of the paths includes the new edge.

a "two-flip tolerant" path search could be used to identify the pairs of paths where exactly one path includes the new edge.

**3.1.1 Two flip tolerant path search.** In a normal directed graph path, only forward edges, i.e. edges that go outward from the current node, are considered. On the other hand, the *two flip path* goes through up to three phases: in the first phase, only backward edges- pointing inward to the current node- are accepted. In the second phase, only forward edges are accepted, and in the third phase, again only backward edges are accepted. As a consequence, there is a node in the path for which both path edges are oriented outwards, that lies between the first and second phase. This will be referred to as the *first flipping point*. Similarly, there is a *second flipping point* between the second and third phase, for which both path edges point inwards.

Consider the situation where a new edge from source node  $S$  to sink  $T$  is being added to a diagram. A two flip tolerant path from  $S$  to  $T$  would correspond to a new pair of parallel paths- flipping point 1 would be the source of both paths, and flipping point 2, the sink. The first path would correspond to the second phase of the flip tolerant path search, and the second path, involving the new edge, would correspond to composing the first phase of the two flip path, the new edge, and the third phase.

To present this idea diagrammatically, we represent path phases with arrows (note that these are the composition of many edges, not a single edge). The new edge is represented with a dashed arrow.



Here,  $(S, F_1, F_2, T)$  is a two flip tolerant path.  $(F_1, S, T, F_2)$  is a new path, created because of the addition of  $(S, T)$ , that conflicts with  $(F_1, F_2)$ .

The two flip tolerant path search returns the set of all paths between a given source and sink that have up to two flips (paths that omit one or more of the three phases are also accepted).

The *path extraction algorithm* then transforms the output of the two flip path search to the set of new pairs of parallel paths that arise in the diagram due to its addition, that need to be verified.

**Path extraction algorithm.** The purpose of this algorithm is to derive the conflicting pairs that a given two-flip tolerant path corresponds to.

Let the new edge added to the diagram be  $(S, T)$ . It has already been shown how each path corresponds to a conflicting pair in the case where a path has two flips. When a path has only the first flip (which is to say, the third phase of the path is missing), but not the second flip, sink node  $T$  can be treated as the sink of the two conflicting path, while the first flipping point remains the source. Similarly when only the second flipping point is present then it is the sink, and  $S$  is the source. Finally when no flipping points are present, there are two possibilities. Either the path is oriented from  $S$  to  $T$ , in which case the conflicting paths are simply the edge  $(S, T)$

and the entire flip-less path, or the path is oriented from T to S. In this case, we have found a cycle that has already been dealt with.

*Theorem.* The result of the two-flip tolerant path search from the source to the sink node of an edge that is to be newly added, followed with the path extraction algorithm has a one-to-one correspondence to the set of new parallel pairs with exactly one path passing through the new edge, with neither paths containing any cycles.

*Proof.* Every element in the output of the path extraction algorithm was by construction a conflicting pair.

It remains to show that every new conflicting pair corresponds to a two flip tolerant path. Let the common source be  $F_1$  and common sink be  $F_2$ .

Only one path passes through (S, T), which we will name path 1. We construct a two flip tolerant path from S to T: phase 1 is the segment of path 1 from  $F_1$  to S, phase 2 is path 2, and phase 3 is the segment of path 1 from T to  $F_2$ . It is possible that some of  $F_1, F_2, S$  and  $T$  coincide, in which case the corresponding segments between them disappear, and the resultant path has fewer than two flips.

We have shown that all paths that need verification are caught by the two flip tolerant search followed with path extraction.

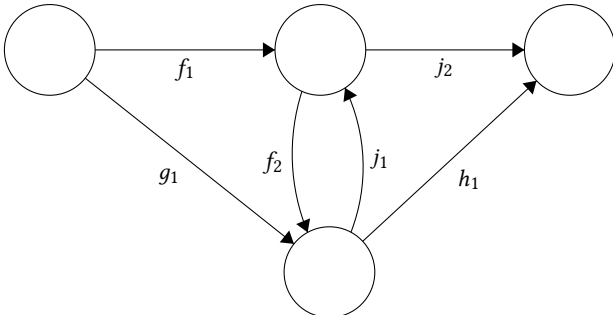
*Implementation.* \*Copy in python code from implementation? Appendix?\*

This algorithm is also in the worst case as bad as factorial in the number of nodes. However, in the average case, it significantly outperforms the naive batch baseline. For sparse diagrams it is even competitive with our ultimate polynomial solution. Average case heuristic analysis will be presented in the evaluation section.

### 3.2 Optimal solution for batch problem

Murota's main result [1] is a solution to the question of whether a given acyclic diagram commutes that returns the minimal number of equality checks. The paper describes the following algorithm to find the ( $V^2E$  bounded) minimal set of pairs that needs to be checked.

A diagram's structure can result in some redundancy, in the sense that verifying a subset of pairs can imply the verification of path pairs that aren't in the subset. For example, in the figure below, verifying  $p_1$  and  $p_2$  implies that the paths in  $p_3$  are equal too.



$$p_1 = (f_1; f_2, g_1)$$

$$p_2 = (j_1; j_2, h_1)$$

$$p_3 = (f_1; h_1, f_1; j_2)$$

The approach in the algorithm, at a high level, is to define a function that takes in a subset of pairs and returns the subset of

pairs whose verification is implied by the verification of the input set. Then, greedy elimination of redundancies is performed until a minimal set is reached.

A bilinking is defined to be a parallel pair that is disjoint but for their terminal nodes. The set of all bilinkings is  $R_0$ . In an acyclic diagram, if all bilinkings are equal, all parallel pairs must also be equal since they any given pair can be expressed as a composition of bilinkings.

Define  $r_1 > r_2$  for bilinkings  $r_1 = \{p_1, q_1\}, r_2 = \{p_2, q_2\} \in R_0$ , if there exists a path  $p$  such that  $\partial p = \partial r_1$  and  $p$  contains  $p_2$ . The  $\langle \rangle$  function is defined as:  $\langle r \rangle = \{s \in R_0 | r > s\}$ .

For bilinking  $s$ , let  $F(s)$  be the vector in  $\mathbb{F}_2^{|E|}$  representing the edges present in  $s$  (there is a dimension corresponding to each edge, the  $n^{th}$  dimension of  $F(s)$  is 1 if the corresponding edge is in  $s$ , and 0 otherwise). Let this function be extended to sets, so that for some set of bilinkings  $S$ ,  $F(S) = \{F(s) | s \in S\}$ .

for a set of bilinkings  $r$ , the closure function  $cl$  is defined as:  $cl(r) = \{s \in R_0 | s \text{ is linearly dependent on } F(r)\}$ .

The closure function on  $r$  basically captures all the pairs that can be made by made by composing or "gluing together" the bilinkings in  $r$ . The aforementioned  $\langle \rangle$  function captures the orientation of paths, and can be used to constrain compositions to be in the right direction.

Using these two function we finally define the function  $\sigma$  on a set of bilinkings  $r$  as  $\sigma(r) = \{s \in R_0 | s \in cl(R \cap \langle s \rangle)\}$ . This is the function used to capture all the pairs whose verification is implied by the verification of pairs in  $r$ .

$\sigma$  is used to iteratively check if a given pair is redundant. bilinkings are eliminated until a minimum "spanning" subset is reached.

An implementation of the described algorithm is

```

Graph existingGraph;
R_s = {}
for each node v in V:
    # Let the subsection of the graph reachable from v be S.
    S = existingGraph.extractReachableSection(v)
    # Create a minimum spanning tree of S
    T = createMinimumSpanningTree(S)
    # Find the excluded edges
    excludedEdges = S.edges - T.edges
    # Use these edges to create bilinkings
    for each edge e in excludedEdges:
        firstPath = T.findPath(source: e.source, sink: e.sink)
        R_s.addElement(new Bilinking(firstPath, e))
return R_s

Find a spanning set Rs, = [r_1, ... , r_k].

R := Rs,
for i:=1 to K do
    if r_i in sigma(R - r_i) then R:=R - r_i;
return R.

sigma(inputSet S, codomain = Rs):
    output = {}
    for bilinking in Rs:
        smallerPairs = {}
        for candidateBilinking in Rs:

```

```

349     if exists path(source: bilinking.source, sink: candidateBilinking.source) and pathInOldGraph = FindPath(
350     exists path(source: candidateBilinking.sink, sink: bilinking.sink): graph: existingGraph,
351     smallerPairs.add(candidateBilinking) sourceNode: S,
352     # Check if there's a linear dependence between the bilinking and smallerPairs sinkNode: T);
353     # vectorize(pair) returns a vector with a dimension for each edge parallelPairs.add((pathInOldGraph, pathWithNewEdge));
354     # with a 1 if pair includes that edge and 0 if it doesn't. catch (pathFindingFailed Exception):
355     matrix = [vectorize(pair) for pair in smallerPairs] // No comparable pairs from node {S} to node {T}
356     matrix = matrix+[vectorize(bilinking)] // that need to be checked
357     # if there is a linear dependence continue
358     if determinant(matrix)%2==0:
359         output.add(bilinking)
360     return output
361

```

The proof of correctness can be found in Murota[1], The number of checks returned by the algorithm is at worst  $O(|V|^2|E|)$ . The overall run time of an optimized implementation is  $O(|V|^4|E|^2)$ .

#### 4 SOLVING THE ONLINE ADDITION PROBLEM

The solution presented here assumes that there is an implicit identity function on every node, as would be the case in most applications. However, a modified version of the algorithm for diagrams without this assumption is presented in the appendix. The modification is similar in spirit to what was described in the description of the naive baselines.

Like with the online baseline, we do not concern ourselves with parallel pairs where neither or both paths pass through the new edge. The key observation that allows us to improve on the online baseline is that for a given (source, sink) pair only a single parallel pair needs to be verified. This is an implication of Theorem 4.1, expanded on later. Theorem 7.2 shows that should our selected set of pairs along with cycles passing through the new edge be verified commutative, the entire diagram must commute. The approach is to identify a parallel pair with exactly one path through the edge for each (source, sink) pair.

Graph existingGraph;  
Edge newEdge;

```

388 Set parallelPairs = new Set();
389 for S in existingGraph.Nodes():
390     for T in existingGraph.Nodes():
391         try:
392             Path pathWithNewEdge =
393             FindPath(
394                 graph: existingGraph,
395                 sourceNode: S,
396                 sinkNode: newEdge.Source()) +
397             newEdge as Path +
398             FindPath(
399                 graph: existingGraph,
400                 sourceNode: newEdge.Sink(),
401                 sinkNode: T)
402             if (S==T):
403                 // Cycles are a special case.
404                 pathInOldGraph = identity(S)
405             else:

```

output parallelPairs;

The try block is executed at most  $O(|V|^2)$  times, so that this is the bound on the number of pairs verified.

The bound is asymptotically tight. This can be seen in the case where the graph contains  $2N$  nodes besides  $S$  and  $T$ . We consider  $N$  of the nodes to be in group 1, and the other  $N$  to be in group 2. Every node in group 1 has a forward node to every node in group 2, as well as to  $S$ .  $T$  has a forward edge to every node in group 2. In this diagram, on the addition of edge  $(S, T)$ ,  $N^2$  paths need to be verified which is polynomial in  $2N+2$ .

Also notice that if trying to optimize for path length (say, if composing functions is expensive) then "find any path" can be replaced with "find shortest path".

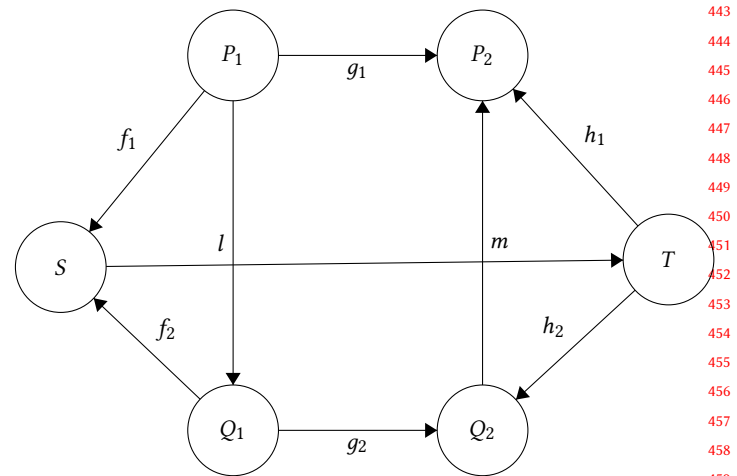
An efficient implementation of the algorithm can run in  $O(|V|(|V|+|E|))$  time at a  $O(|V|^3)$  space cost. In such an implementation, path finding from a given source node to all potential sink nodes would be done in a single  $O(|V| + |E|)$  breadth first search and the results memoized in  $O(|V|^2)$  space.

##### 4.1 Optimization Step

In the case where equality checks are very expensive, we begin by finding the minimal set of (source, sink) pairs such that checking for these pairs logically implies having checked the full diagram.

We observe that there are some redundancies in the diagram.

Consider the following situation:



Each arrow represents a path, and  $(S, T)$  is the new edge being added.

**THEOREM 4.1.** *If conflicting paths  $g_2 = f_2; (S, T); h_2$  then it must be that  $g_1 = f_1; (S, T); h_1$ .*



PROOF. We use the fact that  $f_1=l; f_2$  and  $h_1=h_2; m$ .

$$g_2 = f_2; (S, T); h_2 \Rightarrow l; g_2 = l; f_2; (S, T); h_2$$

$$\Rightarrow l; g_2; m = l; f_2; (S, T); h_2; m \Rightarrow g_1 = f_1; (S, T); h_1$$

□

Note that the proof is not affected if any of these paths is the identity, eg. if  $f_1$  is the identity and  $S$  and  $P_1$  are actually the same node.

We conclude that verifying a comparable pair of paths with end points  $(P_1, P_2)$  implies the verification of all path pairs  $(Q_1, Q_2)$  such that  $Q_1$  is a successor of  $P_1$  and  $P_2$  is a successor of  $Q_2$ . A successor  $S$  to node  $N$  is any node such that there exists a path from  $N$  to  $S$ . Nodes are also their own successors and predecessors.

Under the assumption that the only path operations allowed are composition and replacement of one path by a different, equal path, as would be true when edges are generic functions, and no other information is available, so that  $F$  is a semi-group, this "reduction" rule is also the only rule to reduce the set of path pairs to check by finding implications.

That is to say, if verifying a comparable pair of paths with end points  $(P_1, P_2)$  implies the verification of a pair with endpoints  $(Q_1, Q_2)$ , then it must be that  $Q_1$  is a successor of  $P_1$  and  $P_2$  is a successor of  $Q_2$ .

Using this information it is possible to choose a minimal subset of path pairs to verify.

We construct a graph with a node for each possible (source, sink) pair in the graph- each node then represents a possible choice for parallel pair endpoint pairs, and greedily search for the smallest set of nodes from which the entire graph would be reachable. The idea is to look for "roots" in the graph that have to be included in the ultimate verification set because they have no predecessor and cannot be verified "through" the verification of some other pair. Then all the successors whose verification is implied by the roots are eliminated.

```
# Existing graph
Graph existingGraph;
# New edge
Edge (S, T);
```

```
predecessors = S.predecessors(existingGraph)
successors = T.successors(existingGraph)
```

```
# Construct the graph
Graph terminalPairGraph;
for q in successors:
    for p in predecessors:
        terminalPairGraph.addNode(q, p)
        # addEdge function should handle addition of edges between nodes that
        # haven't been defined yet
        for q_pred in q.predecessors(existingGraph):
            for p_succ in p.successors(existingGraph):
                terminalPairGraph.addEdge((q_pred, p_succ))
```

```
# Now we reduce the graph.
verificationSet = {}
while (len(terminalPairGraph.nodes) > 0):
```

```
# Choose a random node to start reduction from.
current_node = terminalPairGraph.nodes[0]
visited_nodes = set()
# Climb up the graph till a "root" node is found
while (len(currentNode.parents()) > 0):
    visited_nodes.add(currentNode)
    # Choose a parent at random.
    currentNode = currentNode.parents()[0]
    # Deal with cycles by reducing them
    if currentNode in visited_nodes:
        edges = getAllEdges(visited_nodes, terminalPairGraph)
        terminalPairGraph.removeNodes(visitedNodes)
    # All the incoming and outgoing edges from the entire visited
    # transferred to a single node that represents the class
    terminalPairGraph.addNode(currentNode, edges)
    # No predecessor! We should have reached a root!
    verificationSet.add(currentNode)
terminalPairGraph.removeNodes(currentNode.successors(terminalPairGraph))

# This contains the minimal set of (source, sink) pairs whose verification
# implies verification of the entire graph.
return verificationSet
```

The result of the algorithm cannot be further reduced. Also, the verification of the parallel pairs returned in the algorithm implies that the output of the previous algorithm must commute, and transitively with Theorem 7.2 that the entire diagram must commute.

The run time of the first step is  $O(|V|^4)$ , and that of the second step is  $O(|V|)$ , so that the overall bound is  $O(|V|^4)$ .

## 4.2 Verification

Ultimately verification is performed by calling an equality oracle (that is beyond the scope of this work) for every comparable pair returned by the previous stage.

```
for (path_1, path_2) in parallelPairs:
    if (path_1 != path_2):
        print($"Edge addition does not maintain path independence.
        Counterexample: {path_1}, {path_2}")
        return False
print("Edge addition maintains path independence.")
return True
```

## 5 APPLICATIONS

TODO.

### 5.1 Gator

### 5.2 Currency Graph

## 6 EVALUATION

## 7 APPENDIX: THEOREMS AND ADDITIONAL CASES

Consider all the cycles in the graph that pass through the new edge, unique to their terminal point. Define a cycle-pair corresponding to cycle  $c$  to mean the pair of paths  $(c, \text{identity on terminal node of } c)$ .

c). Let  $C$  be the set of all cycle-pairs (unique to their terminal point) corresponding to cycles that pass through the new edge.

Let  $V$ , the set of pairs to verify, be  $C \cup R_0$ .

**LEMMA 7.1.** *A path that involves the new edge more than once is equal to a path without multiple occurrences of the edge, under the assumption that the pairs in  $C$  are verified.*

**PROOF.** Consider the part of the path between the first occurrence of the new edge and the second. This forms a cycle, which has a corresponding check in  $C$  and must be verified to be equal to the identity. This means that the loops containing the multiple occurrences of the new edge may be ignored and the path with a single occurrence of the new edge with the cycles removed is equal to this path.  $\square$

**THEOREM 7.2.** *Verifying that all pairs in  $V$  commute implies that all pairs in  $R_{all}$  commute.*

**PROOF.** Each pair in  $R_{all}$  that is not in  $V$  would fall into one of these categories:

Case A: pairs that do not involve the new edge: By assumption that the original diagram commutes, these pairs must commute.

Case B: Cases where there is a different pair corresponding to this choice of (source, sink) in  $V$  already. There are four paths to be considered, two from each pair. Those paths which do not include the new edge must already be equal by the assumption that the original diagram commutes. Those paths which do include the new edge (S,T) can be reduced, by the case for multiple occurrences of a new edge, to a path with a single occurrence of the new edge as described in the preceding lemma 7.1. These new paths can be divided into three segments: the segment from the source to S, the new edge, and the segment from the last occurrence of T to the sink. The first segments of all paths are equal by assumption that the original diagram commutes. This is also true of the third segment of all the paths. The second segment consists of only the new edge and is the same for all paths. The composition of equal functions is equal, so all the paths passing through the new edge must be equal. Therefore it is sufficient to check one path that passes through the new edge and one that doesn't. Such a pair belongs to  $V$  by construction.  $\square$

Modified algorithm for diagrams without the identity:

```
Graph existingGraph;
```

```
Edge newEdge;
```

```
Set parallelPairs = new Set();
```

```
for S in existingGraph.Nodes():
```

```
    for T in existingGraph.Nodes():
```

```
        try:
```

```
            Path pathWithNewEdge =
```

```
                FindPath(
```

```
                    graph: existingGraph,
```

```
                    sourceNode: S,
```

```
                    sinkNode: newEdge.Source()) +
```

```
                newEdge as Path +
```

```
                FindPath(
```

```
                    graph: existingGraph,
```

```
                    sourceNode: newEdge.Sink()),
```

```
                sinkNode: T)
```

```
            if (S==T):
```

```
                // Cycles are a special case.
```

```
                parallelPairs.add(pathWithNewEdge, pathWithNewEdge+pathWithNewEdge);
```

```
            for edge in (S.forwardEdges - pathWithNewEdge[0]):
```

```
                parallelPairs.add(edge, edge+pathWithNewEdge);
```

```
            for edge in (S.backwardEdges - pathWithNewEdge[-1]):
```

```
                parallelPairs.add(edge, pathWithNewEdge+edge);
```

```
            else:
```

```
                Path pathInOldGraph = FindPath(
```

```
                    graph: existingGraph,
```

```
                    sourceNode: S,
```

```
                    sinkNode: T);
```

```
                parallelPairs.add((pathInOldGraph, pathWithNewEdge));
```

```
            catch (pathFindingFailed Exception):
```

```
                // No comparable pairs from node {S} to node {T}
```

```
                // that need to be checked
```

```
                continue
```

```
cycle = findCycleWithTerminalNode(newEdge.Source()):
```

```
    if cycle:
```

```
        parallelPairs.add(newEdge, cycle+newEdge)
```

```
cycle = findCycleWithTerminalNode(newEdge.Source()):
```

```
    if cycle:
```

```
        parallelPairs.add(newEdge, newEdge+cycle)
```

```
output parallelPairs;
```

The optional minimization algorithm is modified to exclude nodes representing cycles where the source and sink are equal, since the reduction case no longer will apply in the absence of an implicit identity on the terminal node of the cycle.

## 8 CASE STUDIES

## REFERENCES

- [1] Kazuo Murota: HOMOTOPY BASE OF ACYCLIC GRAPHS - A COMBINATORIAL ANALYSIS OF COMMUTATIVE BY MEANS OF PREORDERED MATROID. Discrete Applied Mathematics, vol. 17, Issues 1–2, 135–155 (1987).