Theoretical and Computational Approaches to Determining Sets of Orders for (k, g)-Graphs

Appendix: Algorithms

1 Edge subdivision

```
EdgeDistance := function(graph, edges)
  local distances, pair;

  distances := [];
  for pair in IteratorOfCombinations(edges, 2) do
  Add(distances, Distance(graph, pair[1], pair[2]) + 1);
  od;
  return Minimum(distances);
end;

EdgeSubdivision := function(graph, type)
  local edges, order, numberOfEdges, minimumDistance, combination,
     newVertices, newEdges, newGraph;

  if type = "three-edge-cubic" then
  numberOfEdges := 3;
  minimumDistance := Girth(graph) - 3;
  else
  numberOfEdges := 2;
  minimumDistance := Girth(graph) - 2;
  fi;
  edges := UndirectedEdges(graph);
  order := OrderGraph(graph);
  for combination in IteratorOfCombinations(edges, numberOfEdges) do
  if EdgeDistance(graph, combination) >= minimumDistance then
    if type = "two-edge-cubic" then
    newVertices := 2;
    newEdges := [
      [combination[1, 1], order + 1],
      [combination[1, 2], order + 1],
      [combination[2, 1], order + 2],
      [combination[2, 2], order + 2],
      [order + 1, order + 2]
    ];
```

```
    elif type = "three-edge-cubic" then
    newVertices := 4;
    newEdges := [
      [combination[1, 1], order + 1],
      [combination[1, 2], order + 1],
      [combination[2, 1], order + 2],
      [combination[2, 2], order + 2],
      [combination[3, 1], order + 3],
      [combination[3, 2], order + 3],
      [order + 1, order + 4],
      [order + 2, order + 4],
      [order + 3, order + 4]
    ];
    else
    newVertices := 1;
    newEdges := [
      [combination[1, 1], order + 1],
      [combination[1, 2], order + 1],
      [combination[2, 1], order + 1],
      [combination[2, 2], order + 1]
    ];
    fi;
    Append(newEdges, Filtered(edges, e -> not e in combination));
    Append(newEdges, List(newEdges, e -> Reversed(e)));
    newGraph := EdgeOrbitsGraph(Group(()), newEdges, order +
       newVertices);
    Print(combination, "\n");
    #return newGraph;
  fi;
  od;
  return fail;
end;
```

2 Edge deletion and vertex addition

```
DeleteEdgesAddVertices := function(graph, numberOfEdges,
    numberOfVertices, girth)
  local edges, degree, order, numberOfExtraEdges, deletedEdges,
      filteredEdges,
       newEdges, possibleEdges, extraEdges, newGraph;

  edges := UndirectedEdges(graph);
  degree := VertexDegree(graph, 1);
  order := OrderGraph(graph);
  numberOfExtraEdges := (degree * (OrderGraph(graph) +
     numberOfVertices) / 2) -
              (Size(edges) - numberOfEdges);
  for deletedEdges in IteratorOfCombinations(edges, numberOfEdges) do
    filteredEdges := Filtered(edges, e -> not e in deletedEdges);
    Append(filteredEdges, List(filteredEdges, e -> Reversed(e)));
    possibleEdges := Filtered(
      Combinations(Concatenation(Unique(Flat(deletedEdges)), [order +
          1, order + 2]), 2),
      e -> not e in deletedEdges
    );
    for extraEdges in IteratorOfCombinations(possibleEdges,
       numberOfExtraEdges) do
      newEdges := Concatenation(extraEdges, List(extraEdges, e ->
         Reversed(e)), filteredEdges);
      newGraph := EdgeOrbitsGraph(Group([]), newEdges, order +
         numberOfVertices);
      if Girth(newGraph) = girth and VertexDegrees(newGraph) = [
         degree] then
        return newGraph;
      fi;
    od;
  od;
  return fail;
end;
```

3 Vertex deletion

```
DeleteVertices := function(graph, girth, numberOfVertices)
  local degree, newOrder, vertices, subgraph, edges,
      verticesToReconnect, possibleEdges, numberOfExtraEdges,
      extraEdges, newEdges, newGraph;

  degree := VertexDegree(graph, 1);
  newOrder := OrderGraph(graph) - numberOfVertices;
  for vertices in IteratorOfCombinations(Vertices(graph), newOrder)
     do
    subgraph := InducedSubgraph(graph, vertices);
    edges := UndirectedEdges(subgraph);
    verticesToReconnect := Filtered(Vertices(subgraph),
                                        v -> VertexDegree(subgraph, v
                                           ) <> degree);
    possibleEdges := Combinations(verticesToReconnect, 2);
    numberOfExtraEdges := (degree * newOrder / 2) - Size(edges);
    for extraEdges in IteratorOfCombinations(possibleEdges,
       numberOfExtraEdges) do
      newEdges := Concatenation(edges, extraEdges);
      Append(newEdges, List(newEdges, e -> Reversed(e)));
      newGraph := EdgeOrbitsGraph(Group(()), newEdges, newOrder);
      if Girth(newGraph) = girth and VertexDegrees(newGraph) = [
         degree] then
        return newGraph;
      fi;
    od;
  od;
  return fail;
end;
```

4 Biggs's tree deletion

```
DeleteBiggsTree := function(graph)
  local girth, numberOfLayers, base, layers, removedVertices,
     newVertices, newGraph, vertex, edge;

  girth := Girth(graph);
  numberOfLayers := Int(girth / 4);
  if girth mod 4 in [0, 1] then
    base := [1, Adjacency(graph, 1)[1]];
  else
    base := [1];
    numberOfLayers := numberOfLayers + 1;
  fi;
  layers := Layers(graph, base);
  removedVertices := Concatenation(layers[1..numberOfLayers]);
  newVertices := Filtered(Vertices(graph), v -> not v in
     removedVertices);
  newGraph := InducedSubgraph(graph, newVertices);
  for vertex in layers[numberOfLayers] do
    edge := List(
            Filtered(Adjacency(graph, vertex), v -> not v in
               removedVertices),
            u -> u - Number(removedVertices, x -> x < u)
        );
    AddEdgeOrbit(newGraph, edge);
    AddEdgeOrbit(newGraph, Reversed(edge));
  od;
  return newGraph;
end;
```

5 Circulant graph and its variation

```
ModifyNeighbor := function(vertex, numberOfVertices)
  local neighbor;

  neighbor := vertex mod numberOfVertices;
  if neighbor = 0 then
    return numberOfVertices;
  else
    return neighbor;
  fi;
end;

CirculantGraph4_4 := function(numberOfVertices)
  local edges, vertex, neighbors;

  edges := [];
  for vertex in [1..numberOfVertices] do
    neighbors := [vertex + 1, vertex - 1, vertex + 3, vertex - 3];
    Apply(neighbors, n -> ModifyNeighbor(n, numberOfVertices));
    Append(edges, List(neighbors, n -> [[n, vertex], [vertex, n]]));
  od;
  return EdgeOrbitsGraph(Group(()), Concatenation(edges),
     numberOfVertices);
end;

CirculantGraph4_6 := function(numberOfVertices)
  local edges, vertex, neighbors;

  edges := [];
  for vertex in [1..numberOfVertices] do
    if vertex mod 2 = 0 then
      neighbors := [vertex + 1, vertex - 1, vertex + 7, vertex + 11];
    else
      neighbors := [vertex + 1, vertex - 1, vertex - 7, vertex - 11];
    fi;
    Apply(neighbors, n -> ModifyNeighbor(n, numberOfVertices));
    Append(edges, List(neighbors, n -> [[n, vertex], [vertex, n]]));
  od;
  return EdgeOrbitsGraph(Group(()), Concatenation(edges),
     numberOfVertices);
end;
```