Graph Theory Algorithms and Feynman Diagram Computations

M. Czakon

CAPP05, Zeuthen 3-8 April 2005

I

- Introduction
- Basic definitions
- Graph representations

Feynman diagrams in perturbative calculations

- generation:

  rapid growth of the number of diagrams with the number of loops and legs

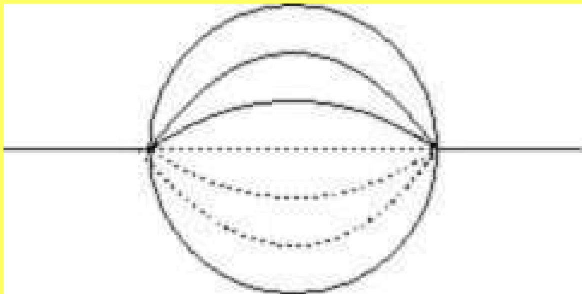
  examples: ~10000 in electroweak 2-loop calculations
  ~50000 in 4-loop beta function calculations

  reasons to use the computer: hard work and errors avoided

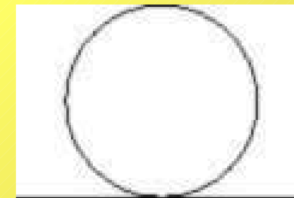
  note: most current problems would be otherwise unsolvable
  note: better methods available for tree-level processes (Dyson-Schwinger equations) which avoid diagrams altogether

- symmetry group determination:

the generation phase should give as little identical diagrams as possible. this gives rise to so-called symmetry factors

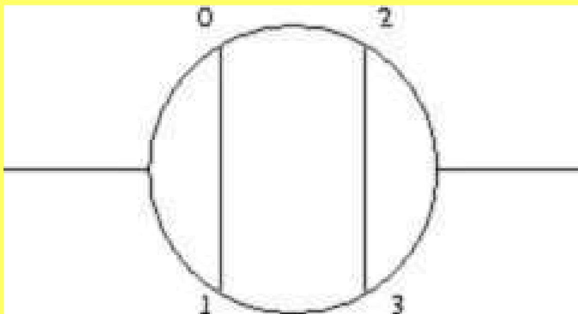


$\sim \frac{1}{n!}$



$\sim \frac{1}{2}$

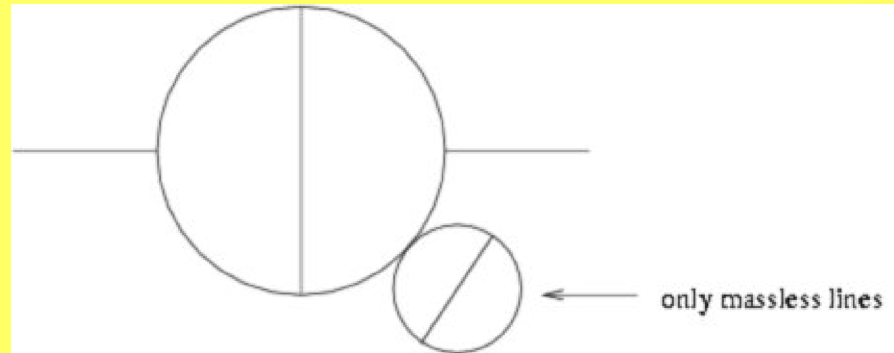
How about this:



there is a symmetry under 0 <-> 1, 2 <-> 3, thus a factor of ½

another use: symmetrization to obtain shorter/faster programs

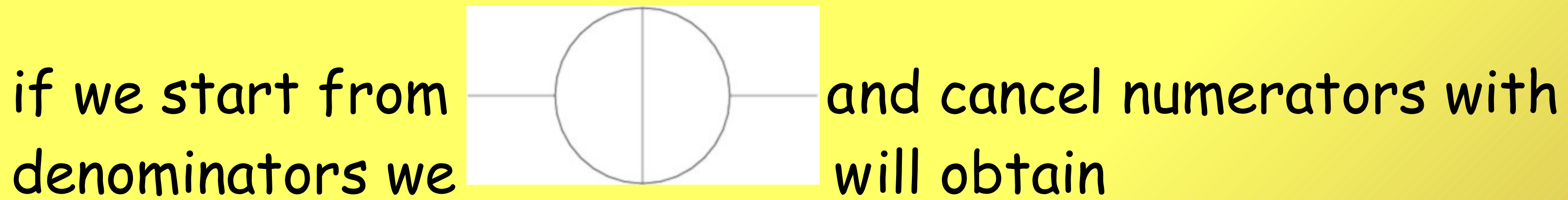
- determination of vanishing diagrams:



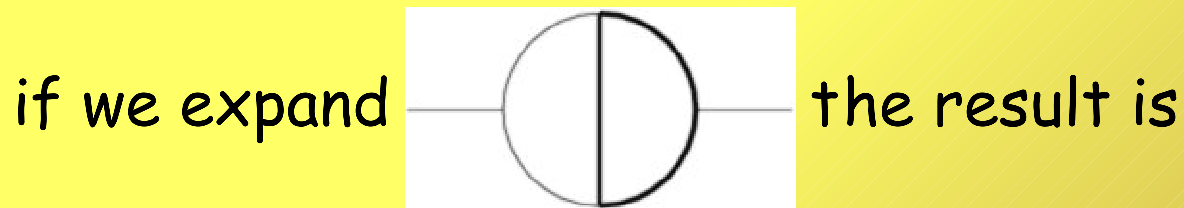
in dimensional regularization this diagram would vanish, because there is a subdiagram, which has no scale.

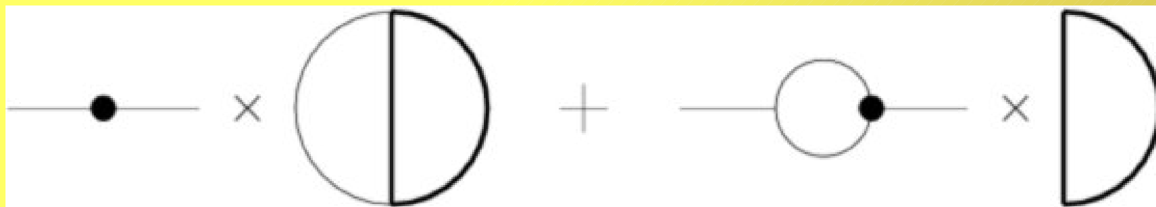
How to check this case at the diagram generation level?

- finding subtopologies:

if we start from  and cancel numerators with denominators we will obtain

Are they equal?

if we expand  the result is



How to organize an expansion automatically?

Major public domain software for diagram manipulation

- qgraf (FORTRAN)

+

fast

-

allows basically for diagram generation and nothing more
not extensible due to code format
the user needs his own output parsers for anything non-trivial

- DIANA (C)

+

extensible
own flexible language

-

interpreter of qgraf output (not really a bad thing)
one more language to learn

- FeynArts (MATHEMATICA)

+

extensible

user friendly

-

slow

doesn't really have a topology generator

topologies hardcoded up to three loops.

- Grace (C)

+

very fast

-

crashes at higher loops and generates too many diagrams

bugs or incorrect algorithm?

A bit of topological analysis can only be found in DIANA

Idea of a C++ library: DiaGen

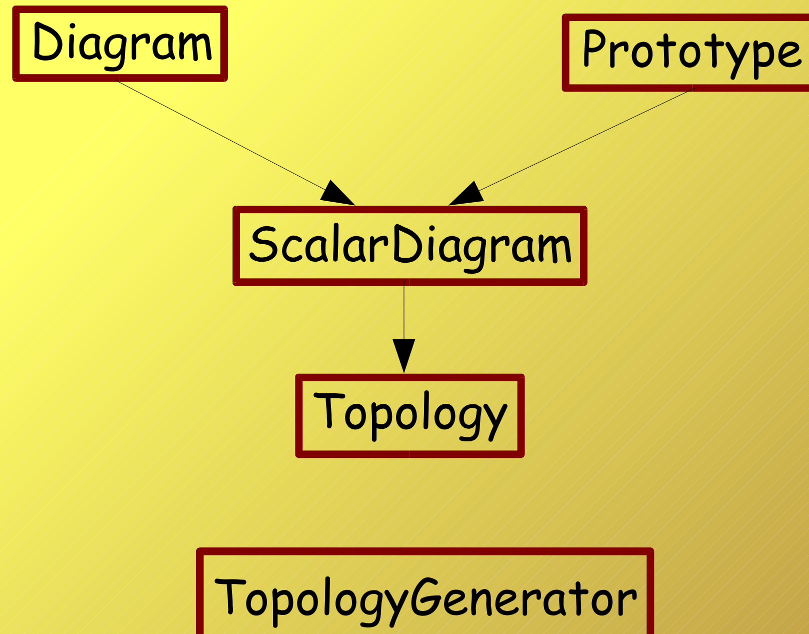
class content:

Field

Vertex

Model

DiagramGenerator



design goals:

speed

functionality

extensibility

Graphs are everywhere:

- electronic circuit design
- route determination in navigation systems
- U-Bahn plans
- pattern matching in computer algebra systems

and of course...

...in Feynman diagram computations!

let's turn to a few:

Basic definitions:

- a graph is a pair $G=(V,E)$ of sets satisfying $E \subseteq V \times V$
- the elements of V are called nodes
- the elements of E are called edges
- a directed graph is a pair (V,E) of disjoint sets with two maps source: E -> V and target: E -> V
- if there are several edges with the same source and target they are called multiple edges
- if the source and target of an edge are equal then the edge is called a loop

Is a QFT topology an undirected or a directed graph?

- It is a directed graph because we want to assign momenta to edges and we need to know the direction of the momentum flow.

- All the other attributes of a topology ignore the direction of the edges.

We need to chose a representation for the V and E sets. The most obvious choice:

- $V = \{0, 1, ..., |V|-1\}$, $E = \{0, 1, ..., |E|-1\}$

fits nicely into the int (or unsigned int) type.

This is not the only possibility. Nodes and edges could be classes.

definition: the degree of a node v w/r to some subset $W \subseteq V$

$$d_G(v,W) = |\{w \in W : \exists e \in E (source(e)=v \wedge target(e)=w) \vee (source(e)=w \wedge target(e)=v)\}|$$

We will also write $d_G(v) = d_G(v,V)$.

How to represent the external lines of a QFT topology?

- take nodes of adjacency 1 as external
- their adjacent edges will correspond to external lines

This may be seen to correspond to introducing sources in the generating functional (path integral)

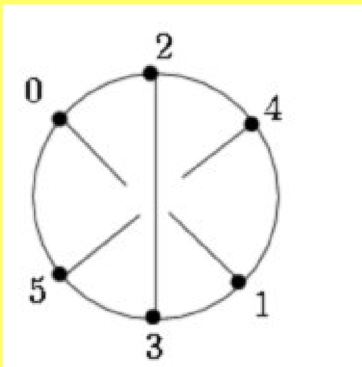
Possible representations of a graph on a computer:

- **Adjacency matrix**: $A_G = (a_{ij})_{0 \le i,j < |V|}$

$$a_{ij} = |\{e \in E : source(e) = i \wedge target(e) = j\}|$$

for undirected graphs it is symmetric
storage needed is of $O(|V|^2)$
most algorithms run in quadratic time w/r to |V|
we will need it only for isomorphism and symmetry group problems

Example:

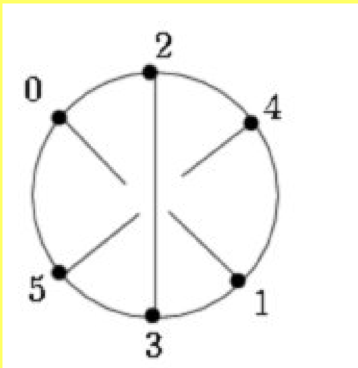


$$A_G = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

- **Adjacency lists**: |V| lists such that the i-th list contains all nodes j for which there exists an edge $e \in E$ that source(e) = i and target(e) = j

  storage needed is of O(|V|+|E|)
  many algorithms can now run in linear time
  additional information will be needed

  Example:



0: 1 2 5
1: 0 3 4
2: 0 3 4
3: 1 2 5
4: 1 2 5
5: 0 3 4

Adjacency lists have to make reference to edges. Suitable representation:

- every node has 2 lists of edges: in-going and out-going

- there is a list of adjacent nodes for each edge

In C++

```
struct AdjacentEdges
{
  vector<int> _edges[2];
};

struct AdjacentNodes
{
  int _node[2];
};

vector<AdjacentEdges> _node;

vector<AdjacentNodes> _edge;
```

Advantage: both sets of edges and nodes are explicit

II

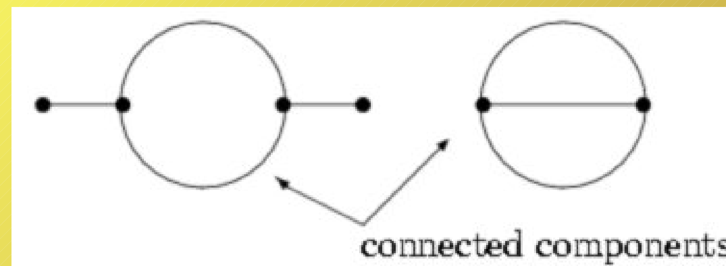
- Connectedness, cycles, shortest paths and graph traversal
- QFT properties: 1PI, on-shell, etc.
- Biconnected components

definition: let (V, E) be a graph. a path from v to w, where $v, w \in V$, is a sequence of nodes $v_0, \ldots, v_k$, such that $v_0 = v$, $v_k = w$ and $(v_i, v_{i+1}) \in E$.

definition: a graph is connected if for every pair v, w of its nodes there is a path from v to w.

definition: a connected component of a graph G=(V,E) is a maximal connected subgraph of G.

Connectedness testing is necessary because the topology generation algorithm generates also disconnected topologies



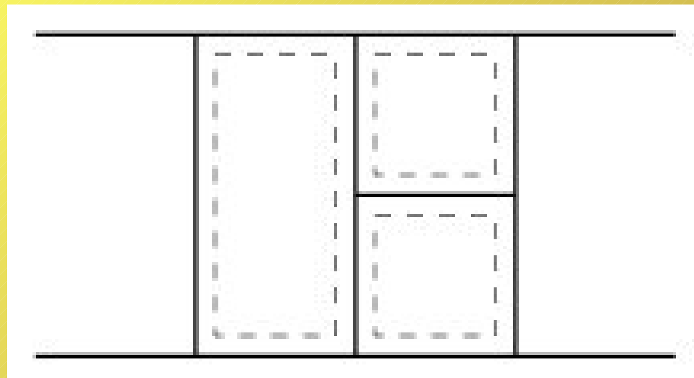
definition: a path from v to w, such that v and w coincide is a cycle

definition: let $\mathbb{N}_2$ be the two element field {0,1}. the edge space E(G) is the vector space of all functions $E \to \mathbb{N}_2$

definition: the cycle space *C*(*G*) is the subspace of E(*G*) spanned by all the cycles of *G*

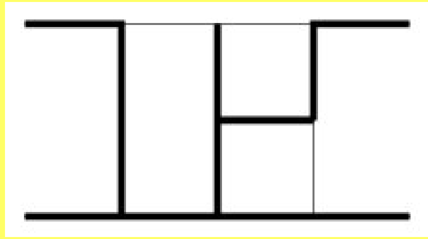
definition: the cyclomatic number is the dimension of *C*(*G*)

the cyclomatic number is simply the number of QFT loops



Proposition: dim C(G) = |E|-|V|+1

Sketch of a proof:



an example graph with cycles
thick lines represent the spanning tree

Any connected tree has |V|-1 edges (easy to see by induction). Edges not contained in the spanning tree must belong to independent cycles, thus dim C(G) = |E|-(|V|-1).

Implication:

QFT topologies have only degree 1, 3 and 4 nodes. Since $2n_e = \sum_i d(v_i) = n_1 + 3n_3 + 4n_4$ and $n_l = n_e - (n_1 + n_3 + n_4) + 1$ we have $n_3 + 2n_4 = n_1 + 2n_l - 2$ and the largest number of nodes is given by the right hand side.

Graph traversal:

- most graph algorithms need to explore the graph starting at some node s

A simple algorithm:

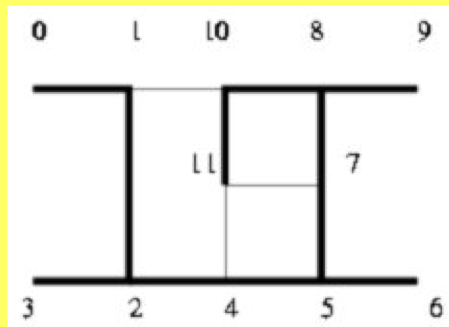
$S \leftarrow \{ s \}$
mark all edges unused
while there are unused edges leaving nodes in S
do chose any $v \in S$ and an unused edge $(v, w) \in E$
    mark (v, w) used
    $S \leftarrow S \cup \{ w \}$
od

Upon termination S contains all nodes reachable from s and is thus the connected component containing node s

A closer look at a realization of the algorithm called
depth first search

At first all the edges are marked unused and the S, T (tree edges) and B (backward edges) sets are empty.

```
procedure dfs(v)
add v to S
for all (v,w) ∈ E
do if (v,w) not used
      then mark (v,w) used
           if w ∉ S
             then add (v,w) to T
                  dfs(w)
             else add (v,w) to B
           fi
   fi
od
end
```



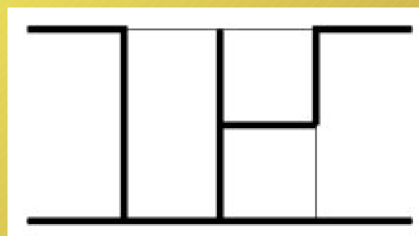
T is the spanning tree

$|B| = \dim C(G)$

if $|S| \neq |V|$, G is not connected

Exercise:

could this tree have been generated by depth first search:



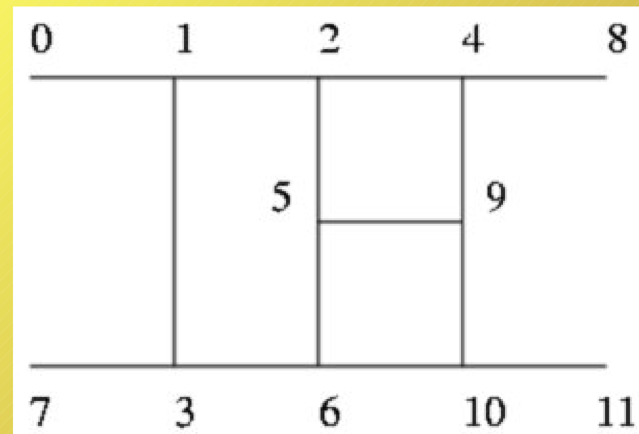
?

if yes, what was the order of the nodes?

In DFS traversal we always start from the last node visited, because the nodes are implicitly on a stack (recursive calls of the procedure)

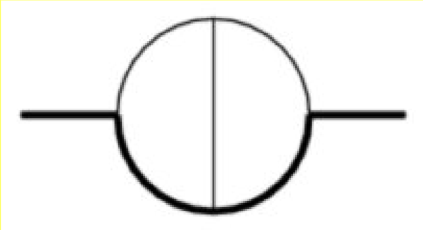
If we replace the stack with a queue, where elements are pushed at the back and popped at the front, the exploration is called **breadth first search**.

```
procedure explorefrom(s)
add s to S
push s on Q
while Q≠Ø
do v←pop(Q)
   for all (v,w)∈E
   do if w∉S
         then add w to S
              push w on Q
      fi
   od
od
```

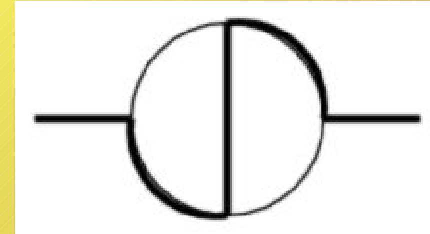


BFS exploration can be used to minimize the length of a path between two nodes. This is the **shortest path** problem.

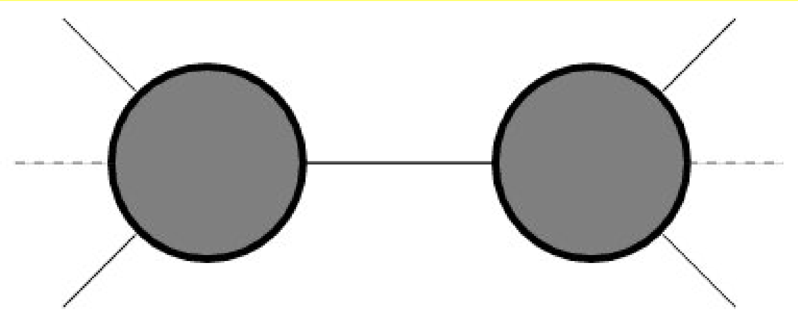
Finding the shortest path may be interesting if we are doing expansions of two-point functions. We want to expand the smallest number of propagators.



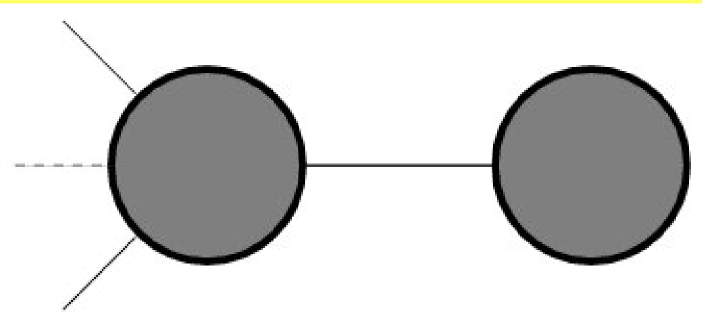
is better than



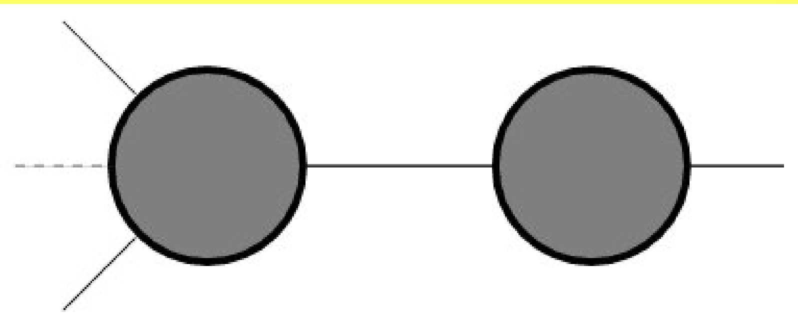
If we can test for connectedness and explore connected components we can easily implement tests for QFT topological properties:



a topology is **1-particle-reducible** if there is at least one internal edge, such that if it removed the graph becomes disconnected



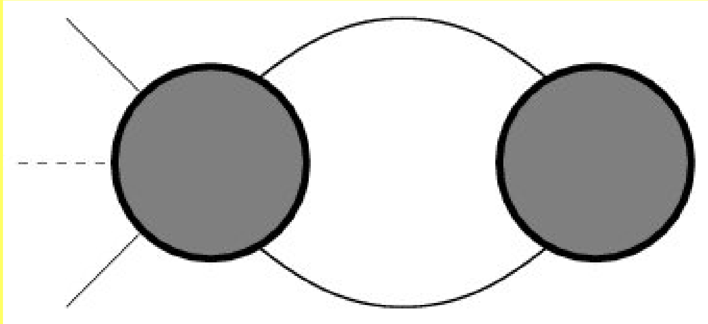
a topology **contains tadpoles** if there is at least one internal edge, such that if it is cut, then one of the connected components contains no external nodes



a topology is not **on-shell** if there is at least one internal edge, such that if it is cut, then one of the connected components contains exactly one external node.

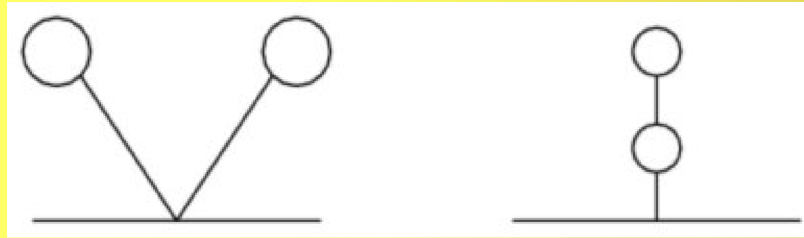
Direct implementation of these definitions is easy, but has high computational complexity. For real Feynman diagrams this is irrelevant

An interesting test: does the topology contain self-energy insertions



cutting two edges gives a component that contains no external nodes

A definition based on momenta alone (two different edges with the same momentum) gives a counter-intuitive result:



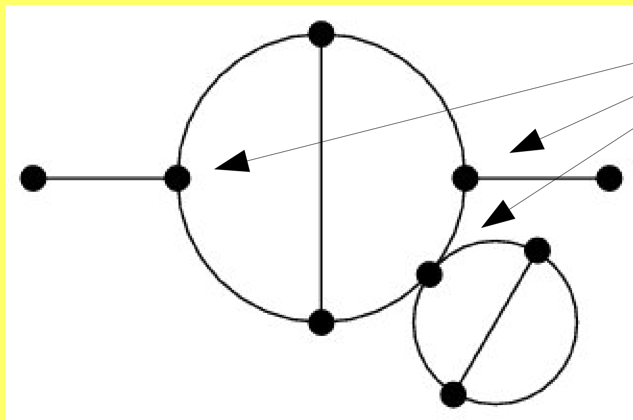
To eliminate the first graph one would have to forbid vanishing momenta, which would also eliminate the second graph

generating topologies with **qgraf** shows that it uses a momentum based definition of a self-energy insertion

definition: a graph G is k-connected if $|G| > k$ and $G-X$ is connected for every set $X \subseteq V$ with $|X|<k$. 2-connected graphs are called biconnected

definition: a node v is an articulation point of G if $G-v$ is not connected

This example illustrates the use of biconnected components:



articulation points

removing these nodes leaves the graph disconnected. there are 4 biconnected components

If we can find all biconnected components and check which have no external momentum flow, we can eliminate scaleless components

The DFS traversal can be adapted to find biconnected components

- apart from the first node, a component is always entered through the articulation point
- backward edges within a component cannot point beneath the articulation point
- let us call the centre of a component the first node in a component explored after the articulation point
- if dfsnum(v) is the number of the node in the dfs tree, then let's define the function lowpt(v), which gives the lowest dfsnum reachable by tree and backward edges
- the center of a component is the node v for which lowpt(v) = dfsnum(parent(v))

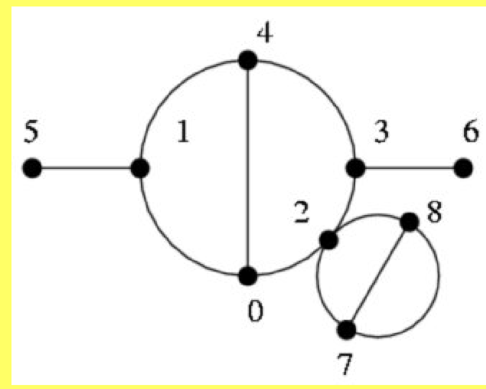
The complete algorithm:

At first, S and unfinished are empty, and count is 0

```
procedure dfs(v)
dfsnum(v)←count; count←count+1
add v to S
lowpt(v)←dfsnum(v)
push v on unfinished
for all (v,w)∈E
do if w∉S
    then parent(w)←v
         dfs(w)
         lowpt(v)←min(lowpt(v), lowpt(w))
    else lowpt(v)←min(lowpt(v), dfsnum(w))
    fi
od
if dfsnum(v) > 0 and lowpt(v)=dfsnum(parent(v))
then repeat w←pop(unfinished)
      until w=v co the nodes popped together with the
                   the node parent(v) form the b.c.c.
fi
end
```

```
#include <iterator>
#include "Topology.hpp"

using std::ostream_iterator;
using std::cout;
using std::endl;

int
main()
{
  Topology t(9);

  t.insert_edge(5,1);
  t.insert_edge(1,4);
  t.insert_edge(1,0);
  t.insert_edge(4,0);
  t.insert_edge(4,3);
  t.insert_edge(0,2);
  t.insert_edge(2,3);
  t.insert_edge(3,6);
  t.insert_edge(2,7);
  t.insert_edge(2,8);
  t.insert_edge(7,8);
  t.insert_edge(7,8);

  t.postscript_print("con.ps");

  vector<TopologyComponent> components = t.biconnected_components();

  for (vector<TopologyComponent>::iterator c = components.begin();
       c != components.end(); ++c)
    {
      cout << "nodes: ";
      copy(c->_nodes.begin(), c->_nodes.end(),
           ostream_iterator<int>(cout, " "));
      if (c->_vacuum) cout << ", vacuum";
      cout << endl;
    }
}
```

Code for



If compiled with:

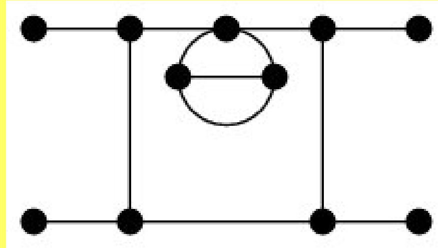
```
g++ -O -o con con.cpp Topology.cpp
./con
```

The output is:

```
nodes: 5 1
nodes: 8 7 2 , vacuum
nodes: 6 3
nodes: 2 3 4 1 0
```

Exercise:

modify the code to determine the b.c.c. of (chose some labelling)



print also the edges and articulation points (see Topology.hpp)

III

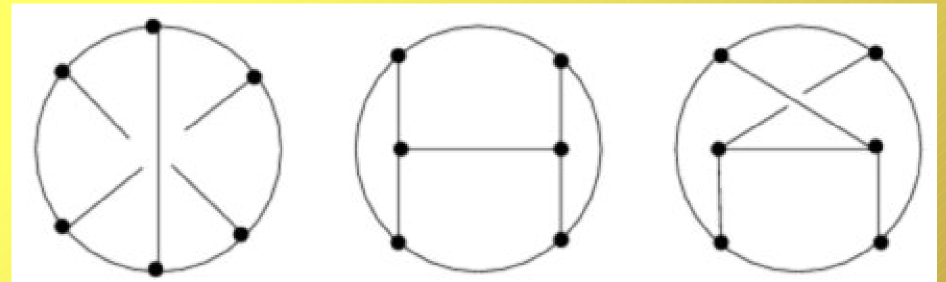
- Graph isomorphism
- Symmetry groups

definition: two graphs (V,E) and (V',E') are isomorphic if there is a bijection σ: V -> V' such that σ(E) = E'.

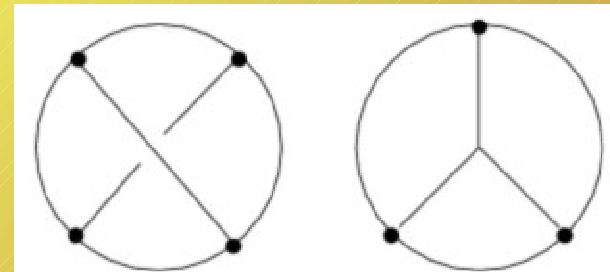
We need the concept of graph isomorphism when generating topologies:

calculating diagrams is expensive, we don't want to calculate the same thing many times

Are these isomorphic?



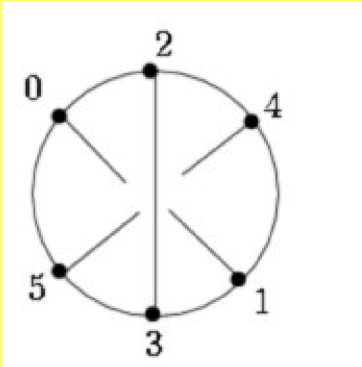
What about these?



Graph isomorphism has other surprising applications:

```
id f(x?,x?)*g(y?,z?)*g(z?,y?) = 1;
```

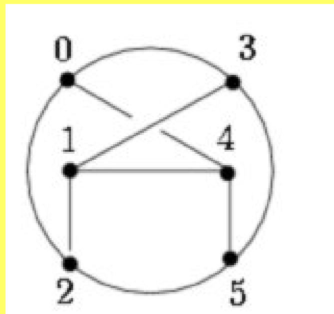
Let's check:



Edge list: (0,1)(2,3)(4,5)(1,3)(3,5)(5,0)(0,2)(2,4)(4,1)

Adjacency matrix:

0 1 1 0 0 1
1 0 0 1 1 0
1 0 0 1 1 0
0 1 1 0 0 1
0 1 1 0 0 1
1 0 0 1 1 0



Edge list: (0,4)(4,5)(5,2)(2,1)(1,3)(3,0)(1,4)(0,2)(3,5)

Adjacency matrix:

0 0 1 1 1 0
0 0 1 1 1 0
1 1 0 0 0 1
1 1 0 0 0 1
1 1 0 0 0 1
0 0 1 1 1 0

They don't look the same...

Let's try with a program:

```
#include "Topology.hpp"

using std::cout;
using std::endl;

int
main()
{
  Topology t1(6), t2(6);

  t1.insert_edge(0,1);
  t1.insert_edge(2,3);
  t1.insert_edge(4,5);
  t1.insert_edge(1,3);
  t1.insert_edge(3,5);
  t1.insert_edge(5,0);
  t1.insert_edge(0,2);
  t1.insert_edge(2,4);
  t1.insert_edge(4,1);

  t2.insert_edge(0,4);
  t2.insert_edge(4,5);
  t2.insert_edge(5,2);
  t2.insert_edge(2,1);
  t2.insert_edge(1,3);
  t2.insert_edge(3,0);
  t2.insert_edge(1,4);
  t2.insert_edge(0,2);
  t2.insert_edge(3,5);

  cout << t1.node_labelling();

  cout << t2.node_labelling();

  if (isomorphic(t1,t2))
    cout << "isomorphic" << endl;
  else
    cout << "not isomorphic" << endl;
}
```

If we compile this program with:

```
g++ -O -o iso iso.cpp Topology.cpp; ./iso
```

The output is:

```
0 3 4 1 2 5
0 1 5 2 3 4
isomorphic
```

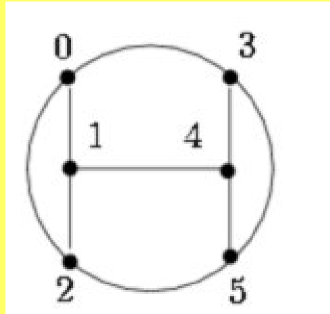
With the node permutations above both matrices are:

```
0 0 0 1 1 1
0 0 0 1 1 1
0 0 0 1 1 1
1 1 1 0 0 0
1 1 1 0 0 0
1 1 1 0 0 0
```

The bijection is therefore:

0->0, 1->2, 2->3, 3->1, 4->5, 5->4

A similar program gives for:



the adjacency matrix:

0 0 0 1 1 1
0 0 1 0 1 1
0 1 0 1 0 1
1 0 1 0 1 0
1 1 0 1 0 0
1 1 1 0 0 0

It is obviously not isomorphic with the two others

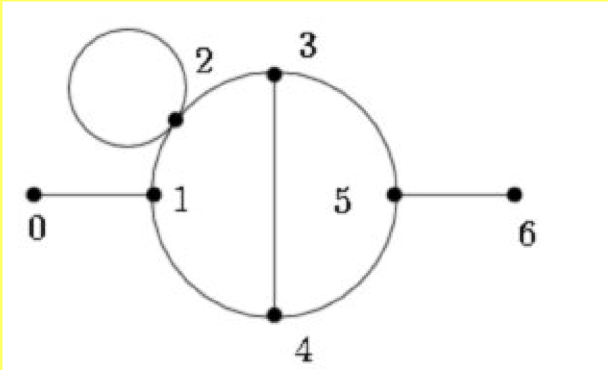
How to construct an algorithm for isomorphism testing?

- First idea: transform both matrices through all possible permutations and compare. This would make 6!*6! = 518400 cases to consider
- Better: transform every matrix individually to obtain the smallest representation w/r to lexicographic ordering. Only 2*6! = 1440 cases to consider. 360 times faster!
- The algorithm used by the Topology class needs only 142 cases and is thus still more than 10 times faster!

First, we need some definitions:

- a partition of a set V is a set of disjoint non-empty subsets of V whose union is V.
- an element of a partition is called a cell.
- we say that π_1 is finer than π_2 if every cell of π_1 is a subset of some cell of π_1. we write $\pi_1 \leq \pi_{2.}$ under the same conditions π_2 is coarser than π_1.
- a partition π is equitable if for every $V_1, V_2 \in \pi$, $d(v_{1,} V_1) = d(v_{2,} V_2)$ for all $v_i \in V_i$.
- $\xi(\pi)$ denotes the unique coarsest partition finer than π.

An example:



The coarsest partition is always trivial $\pi_0 = \{0,1,2,3,4,5,6\}$

A partition into nodes of the same degree is $\pi_1 = \{\{0,6\},\{1,3,4,5\},\{2\}\}$

The finest partition is always discrete $\pi_2 = \{\{0\},\{1\},\{2\},\{3\},\{4\},\{5\},\{6\}\}$

Exercise: Convince yourself that $\xi(\pi_0) = \xi(\pi_1) = \pi_2$

It is clear that we will not obtain the same graph if we permute nodes from different cells of an equitable partition.

If we can define an ordering of cells equitable partitions may reduce the number of permutations in isomorphism testing!

Algorithm : Given a graph $G=(V, E)$, compute $R(G, \pi, \alpha) \in \underline{\Pi}(V)$ (the set of ordered paritions of V),
where $\pi \in \underline{\Pi}(V)$ and $\alpha=(W_{1,} \dots, W_M) \subseteq \pi$.

(1) $\overline{\pi} := \pi$
$m := 1$

(2) if ($\overline{\pi}$ is discrete or $m > M$) stop : $R(G, \pi, \alpha) = \overline{\pi}$
$W := W_m$
$m := m+1$
$k := 1$
$\{$suppose $\overline{\pi} = (V_{1,} \dots, V_r)$ at this point$\}$

(3) define $(X_{1,} \dots, X_s) \in \underline{\Pi}(V)$ such that for any $x \in X_i$, $y \in X_j$, $d(x, W) < d(y, W)$ iff $i < j$.
if $(s=1)$ go to (4)
let t be the smallest integer such that $|X_t|$ is maximum $(1 \leq t \leq s)$
if ($W_j = V_k$ for some j $(m \leq j \leq M)$) $W_j := X_t$
for $1 \leq i < t$ set $W_{M+i} := X_i$
for $t < i \leq s$ set $W_{M+i-1} := X_i$
$M := M+s-1$
update $\overline{\pi}$ by replacing the cell V_k with the cells $X_{1,} \dots, X_s$ in that order

(4) $k := k+1$
if $(k \leq r)$ go to (3)
go to (2)

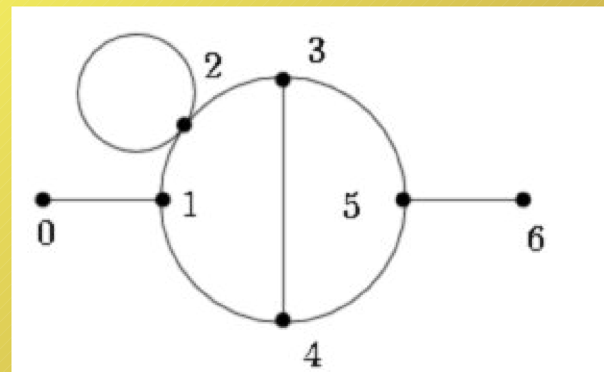
$R(G, \pi, \alpha)$ generates $\xi(\pi)$ in two cases:

(1) $\alpha = \pi$

(2) if there is some equitable partition π' which is coarser than π and $\alpha \subseteq \pi$ such that for any $W \in \pi'$, we have $X \subseteq W$ for at most one $X \in \pi \setminus \alpha$

In our example graph the algorithm generates:

({0, 1, 2, 3, 4, 5, 6})
({0, 6}, {1, 3, 4, 5}, {2})
({0, 6}, {3, 4}, {1, 5}, {2})
({0, 6}, {4}, {3}, {1, 5}, {2})
({0, 6}, {4}, {3}, {5}, {1}, {2})
({0}, {6}, {4}, {3}, {5}, {1}, {2})



The idea of the algorithm determining a unique representation for the adjacency matrix is:

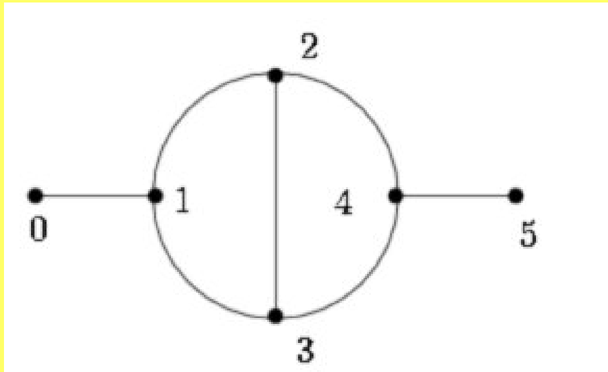
- find the equitable partition starting from the trivial partition
- if it is discrete then there is nothing more to do
- if not permute the nodes within the cells and look for the minimal adjacency matrix

The last step can be further improved:

- at every step, we chose a cell and select a node to be placed before the nodes from the selected cell.
- we find the equitable partition
- if this partition is discrete, we have an allowed permutation

This procedure generates a tree of partitions (search tree)

An example with a non-trivial search tree:



The equitable partition is:
({0, 5}, {2, 3}, {1, 4})

whereas the generated allowed permutations:

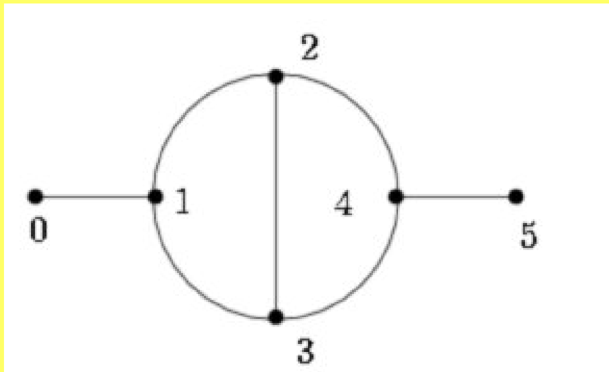
0 5 3 2 4 1
5 0 2 3 1 4
5 0 3 2 1 4
0 5 2 3 4 1

It may also be interesting to compare topologies without permuting the external nodes.

This can be done by taking a starting partition in which every external node is in a single cell.

The topology class does this if we call the method
Topology::fix_external_nodes()

The same algorithms can be used to determine the symmetry group. We use the permutations of the search tree and check whether the adjacency matrix is left invariant. In our case:



Topology::node_symmetry_group():

0 1 2 3 4 5
0 1 3 2 4 5
5 4 2 3 1 0
5 4 3 2 1 0

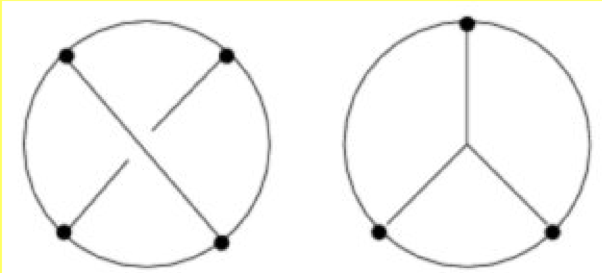
Or the internal symmetry subgroup, when the external nodes are not permuted:

Topology::internal_node_symmetry_group():

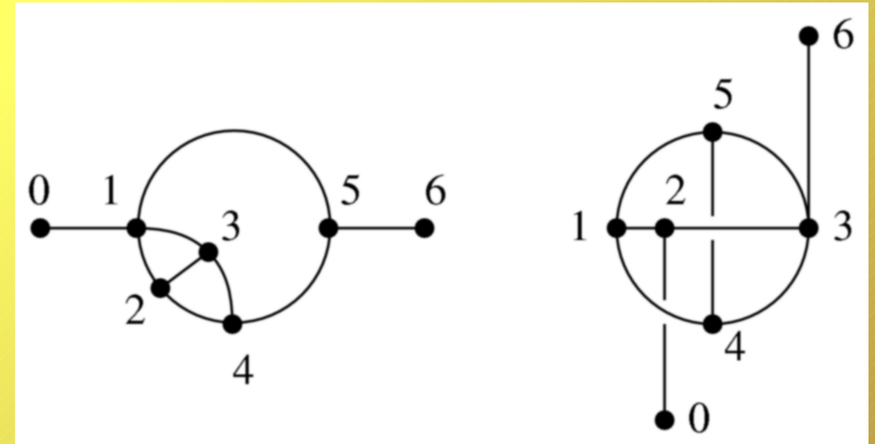
0 1 2 3 4 5
0 1 3 2 4 5

Exercises:

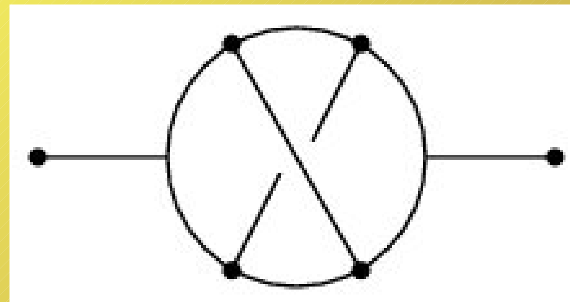
chose a node labelling and suitably modify the program iso.cpp for



are these graphs isomorphic?
what if permutations of external nodes were not allowed?



find the symmetry and internal symmetry groups for



IV

- Graph generation
- Graphs and integrals

Basic ideas behind graph generation:

- generate adjacency matrices.
  the direction and ordering of edges is irrelevant.
- use the relation $n_3 + 2n_4 = n_1 + 2n_l - 2$ and start from nodes of degree 1 and 3.
  (an arbitrary node partition is also possible)
- allow for a specified number of nodes of degree 2.
  these will serve as propagator counterterm insertions.
- unless otherwise specified, reject disconnected graphs.
- insert the generated topologies into a set (fixing by default the external nodes).
  The C++ structure `set<Topology>` will reject isomorphic graphs.

Some storage consideration:

- in this basic algorithm storage requirements grow with the number of topologies in the output

- **an example:**

  6-loop 2-point function topologies with equivalent external nodes but without rejection of 1-particle-reducible cases

  899575 topologies generated
  ~500 MB storage used
  ~6 min (2 GHz Xeon machine partly occupied)

- **conclusion:** hitting limits of storage capacity occurs always later than hitting limits of the ability to subsequently evaluate the diagrams

Where is the problem?

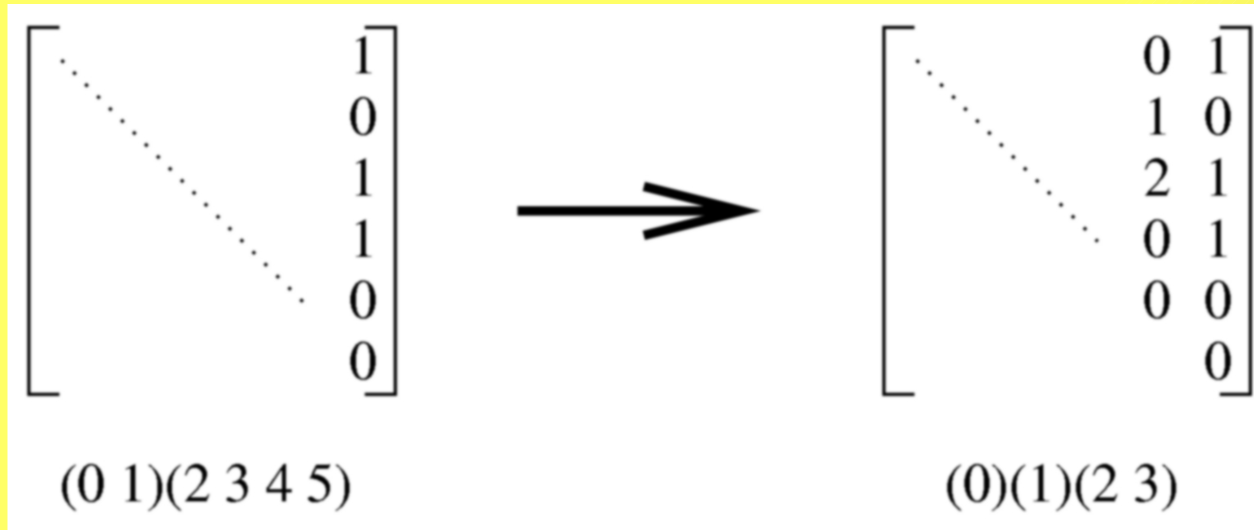
- if we want to generate 4-loop 2-point function topologies the maximum number of nodes is 10 (8 nodes of degree 3)
- there will be raughly 10! = 3628800 different matrices for any graph
- as an improvement we can forbid permutations of nodes of different degree by fixing their numbering within the adjacency matrix. we would still have a slowdown factor of 2!*8! = 80640

What would happen in the 7-loop ϕ^4 theory?

- the vertex would have 4 degree 1 nodes and 8 degree 4 nodes, thus the slowdown factor of 4!*8! = 967680

First improvement: lexicographic ordering

- assign degrees to the nodes in ascending order
- define a partition P_{n-1} of the nodes, such that cells of P_{n-1} contain nodes of the same degree
- fill the adjacency matrix starting from the last node
- sort into descending order the elements of a column j of the adjacency matrix corresponding to nodes within a single cell of P_j
- define P_{j-1} by splitting cells of P_j to contain nodes that have the same adjacency in the column j



Example: 3-loop 2-point function, no degree 4 nodes

- 49 topologies
- naïve estimate of the number of all matrices: 2!*6!*49 = 70560
- generated in reality: 36360
- with sorting: 801
- speedup: ~45

Second improvement:

- suppose that for a given node j there is a node k, with k > j, such that k and j belong to the same cell of P_k .
  if the transposition of j and k would generate a lexicographically larger matrix, then the current matrix is rejected.

- note: the relation $C \subset V \times V$, such that $(j,k) \in C$ if node j should be transposed with node k is a directed graph, the comparison graph

- this algorithm would generate different matrices only

- we simplify the test by checking only whether the column k after transposition is not greater than before transposition

$$\begin{bmatrix} \ddots & 0 & 1 & \\ & 2 & 0 & \\ & 0 & 2 & \\ & 2 & 0 & \\ & \vdots & \vdots & \ddots \end{bmatrix}$$

$(0\ 1)(2\ 3)$

Example: 3-loop 2-point function, no degree 4 nodes

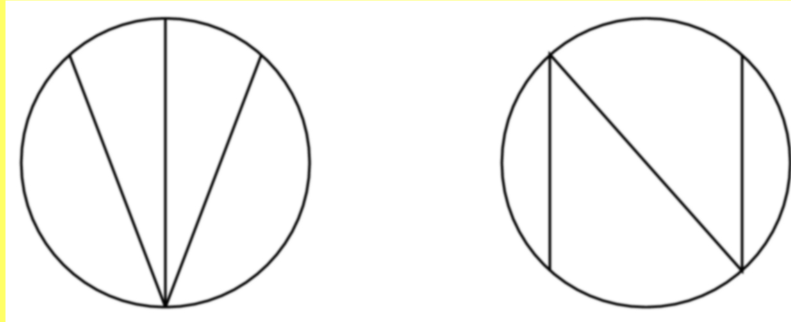
- 49 topologies
- naïve estimate of the number of all matrices: 2!*6!*49 = 70560
- generated in reality: 36360
- with sorting: 801
- with comparison: 61
- speedup: ~596

Last thing to improve: reduce the number of disconnected graphs

```
#include <sstream>
#include "TopologyGenerator.hpp"

using std::ostringstream;
using std::cout;
using std::endl;

int
main()
{
  int count = 0;

#if 1
  TopologyGenerator generator(2, 2, OneParticleIrreducible);
#else
  vector<int> node_count(4);
  node_count[0] = 0;
  node_count[1] = 0;
  node_count[2] = 4;
  node_count[3] = 0;

  TopologyGenerator generator(node_count, EquivalentExternalNodes | OneParticleIrreducible | NoSelfEnergies);
#endif

  while (generator.next_topology())
    {
      ++count;

      Topology t = generator.current_topology();
      t.assign_momenta();
      t.print_edge_list();
      cout << endl;

      ostringstream name;
      name << "top" << count << ".ps";
      t.postscript_print(name.str());
    }
  cout << count << " generated topologies" << endl;
}
```

Graph isomorphism vs integral equality

- Are these two isomorphic? Are the associated integrals equal?



- Graph isomorphism testing is not always sufficient to decide of integral equality

- A different algorithm is needed

- Basic idea: use momentum distributions

- Efficiency requires a non-trivial algorithm to be found in the Prototype class.

Literature:

- R. Diestel, "Graph Theory", Springer-Verlag, New York, 2000, http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/index.html
- K. Mehlhorn, "Data Structures and Efficient Algorithms", Springer-Verlag, EATCS Monographs, 1984, http://www.mpi-sb.mpg.de/~mehlhorn/DatAlgbooks.html
- B. D. McKay, "Practical Graph Isomorphism", Congressus Numerantium **30** (1981) 45, http://cs.anu.edu.au/~bdm/nauty/

Exercises:

- how many graphs does one need to calculate the 7-loop β-function of a ϕ^4 theory? (2- and 4-point functions, 1PI and equivalent external nodes)

- how many master 4-loop vacuum topologies are there? (only degree 3 nodes, no self-energy insertions, 1PI)

- how many tree-level topologies with 8 external nodes are there, when the external nodes are equivalent? what happens if they are equal?