Parallelizing zigzag persistence

Mikael Vejdemo-Johansson

Royal Institute of Technology
Stockholm, Sweden

KTH Computer Science
and Communication

16 October 2012

Outline

Persistent homology

Classical persistence

A diagram of spaces

$$X_0 \hookrightarrow X_1 \hookrightarrow \ldots \hookrightarrow X_n$$

produces a diagram of homology groups

$$H_*X_0 \to H_*X_1 \to \cdots \to H_*X_n$$

Decomposes (over a field) into *intervals* of 1-dimensional spaces with identity maps.

Zigzag persistence

But what if spaces do not include cleanly? What if simplices appear and disappear?

Zigzag homology

Diagram of Dynkin type A_n; arrows can take arbitrary directions.

$$X_0 \hookrightarrow X_1 \hookleftarrow \ldots \hookrightarrow X_n$$

produces

$$H_*X_0 \to H_*X_1 \leftarrow \cdots \to H_*X_n$$

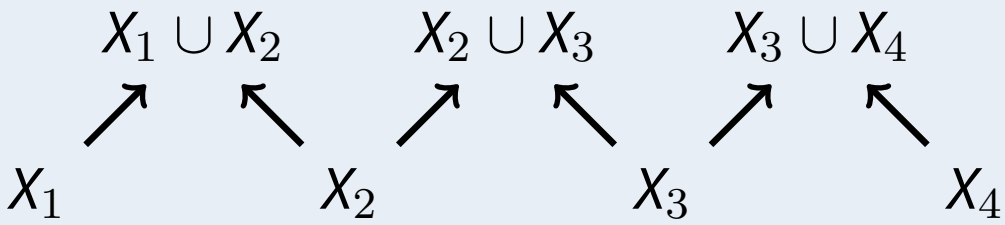
Gabriel (1970) proved that these, too, decompose into intervals of 1-dimensional spaces with identity maps.

Union zigzag

Carlsson & de Silva introduced zigzag persistence, and described a *bootstrap* type technique:

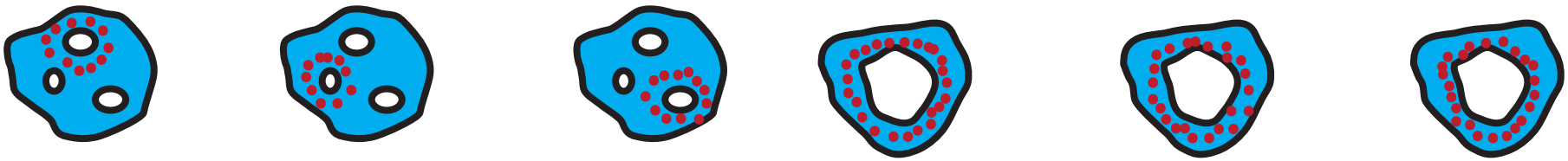
Union zigzag

Suppose $X_1, \ldots, X_n$ are approximations of a space X. The zigzag diagram:

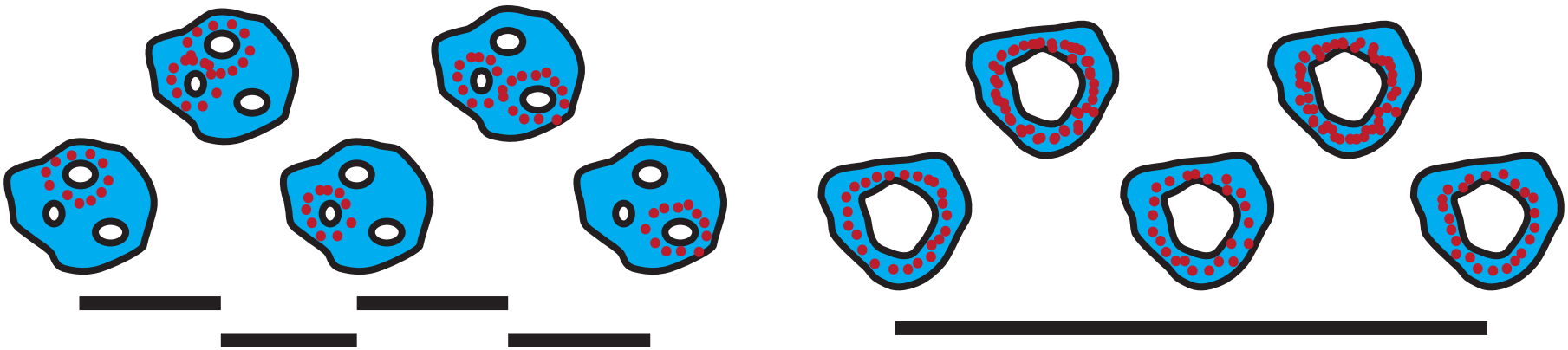


gives rise to a topological style bootstrap: topological features that are artifacts of the approximations live locally; features that are in X exist everywhere.

Bootstrap example



Bootstrap example



Previous zigzag algorithm

Morozov already presented a parallelizable but different algorithm for computing zigzag persistent homology.

Basic formulation

Examines the act of adding and removing a simplex, demonstrates how these primitive operations influence the results.

Advanced formulation

This can be recast into essentially a matrix multiplication, on the order of # insertions and deletions of simplices.

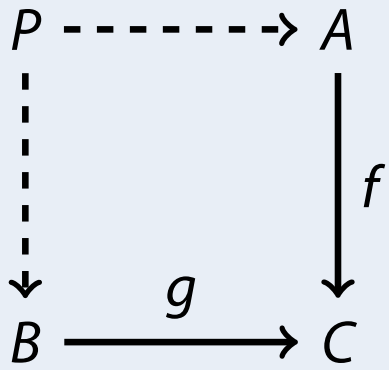
For the bootstrap case, this is prohibitively expensive. Other use cases, this is close to optimal.

Outline

Pullbacks of linear maps

Pullbacks in category theory

The pullback P of two maps $A \xrightarrow{f} C \xleftarrow{g} B$ is a least specific object with maps to A and B such that the diagram



commutes.

Pullbacks in vector spaces

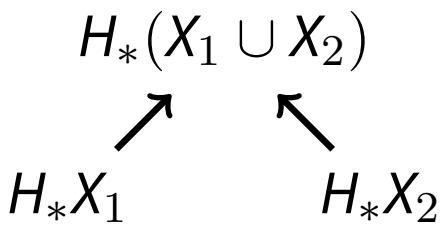
The pullback has a particularly easy formulation in the category of vector spaces:
P is the subspace of $A \oplus B$ where the maps agree:

$$P = \{(a, b) \in A \oplus B : fa = gb\}$$

Pullbacks can compute zigzags

The key recognition here is that the pullback computes the subspace where functions agree.

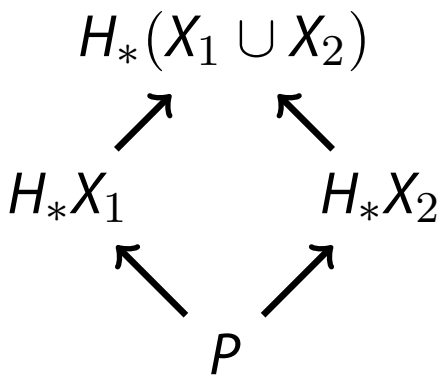
Consider the diagram, where all maps are induced from inclusions:



Pullbacks can compute zigzags

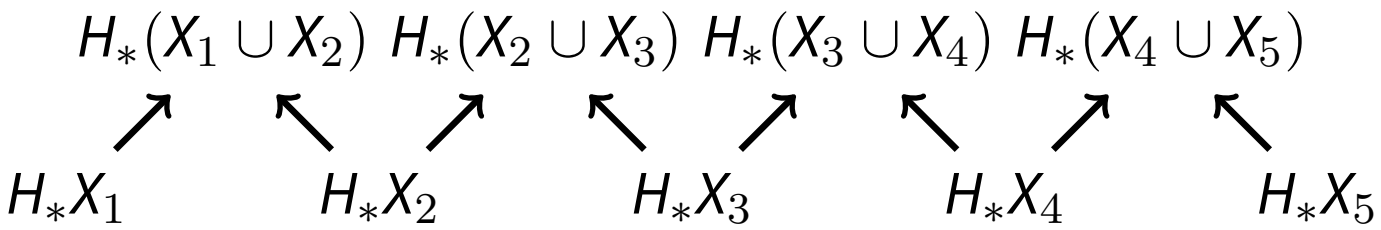
The key recognition here is that the pullback computes the subspace where functions agree.

Consider the diagram, where all maps are induced from inclusions:

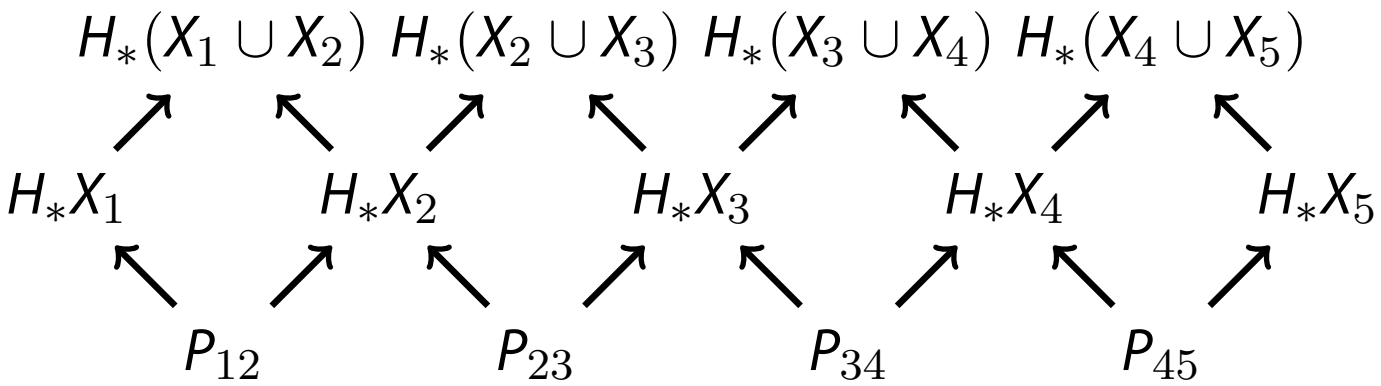


Basis vectors of P encode classes in H_*X_1 and H_*X_2 that agree, up to homology, in $X_1 \cup X_2$.

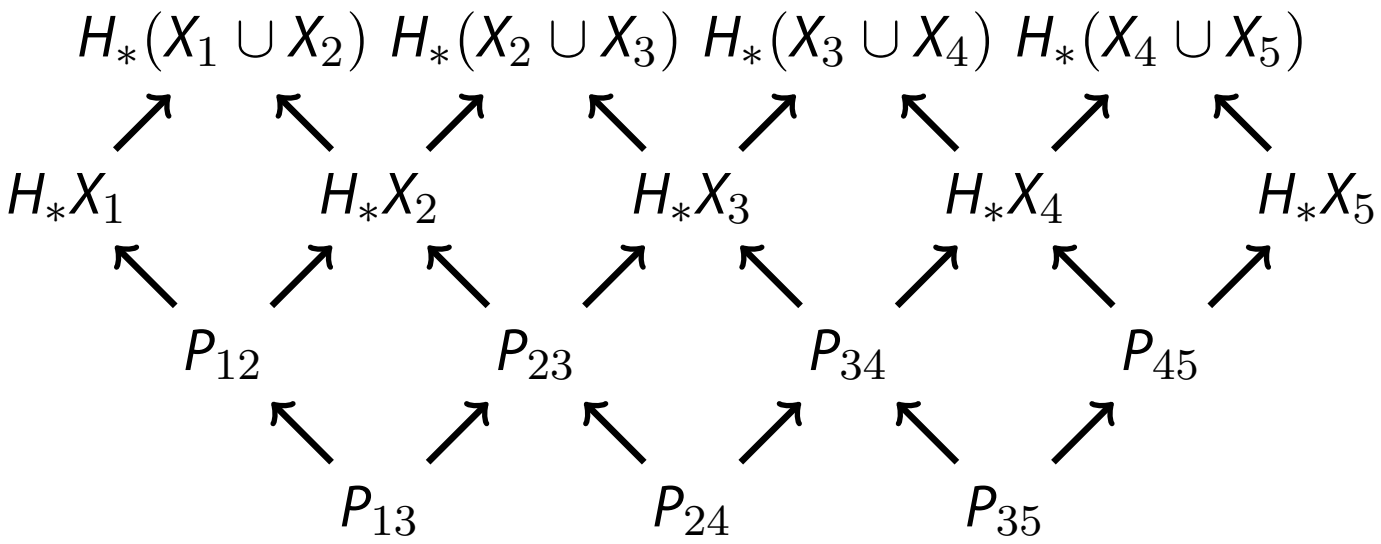
Algorithm overview



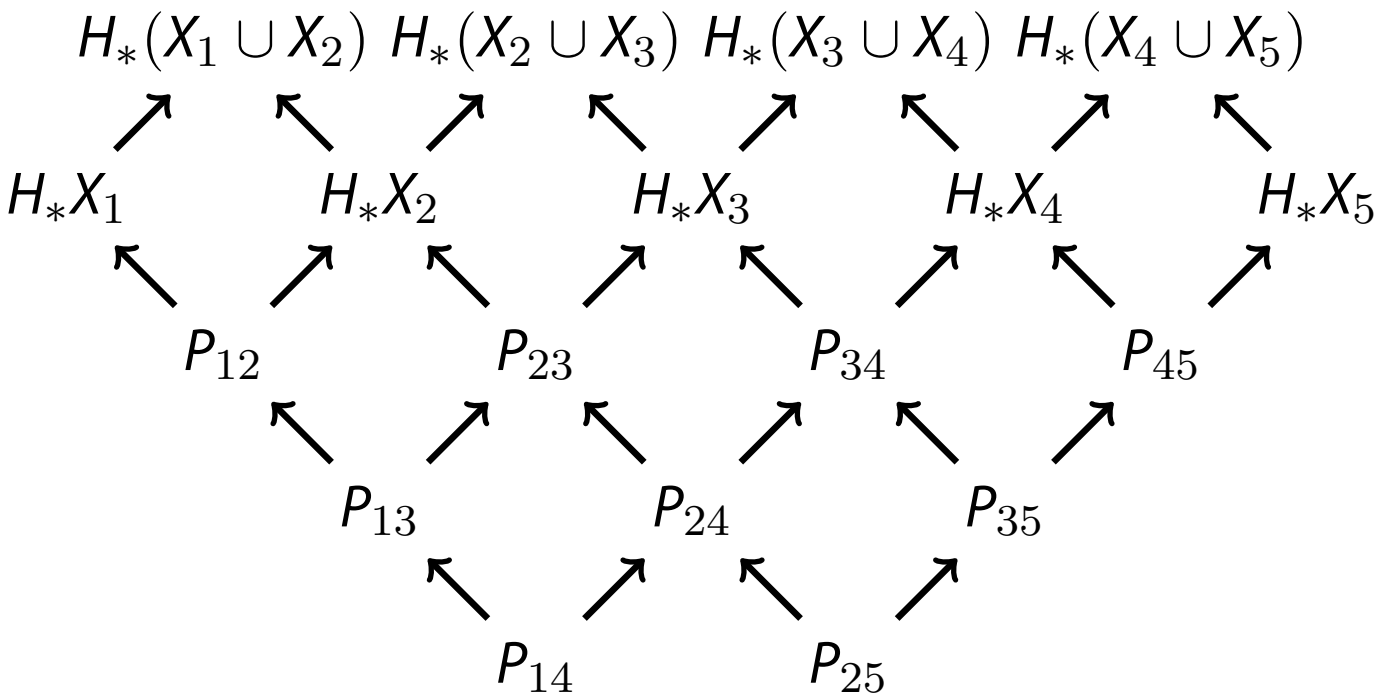
Algorithm overview



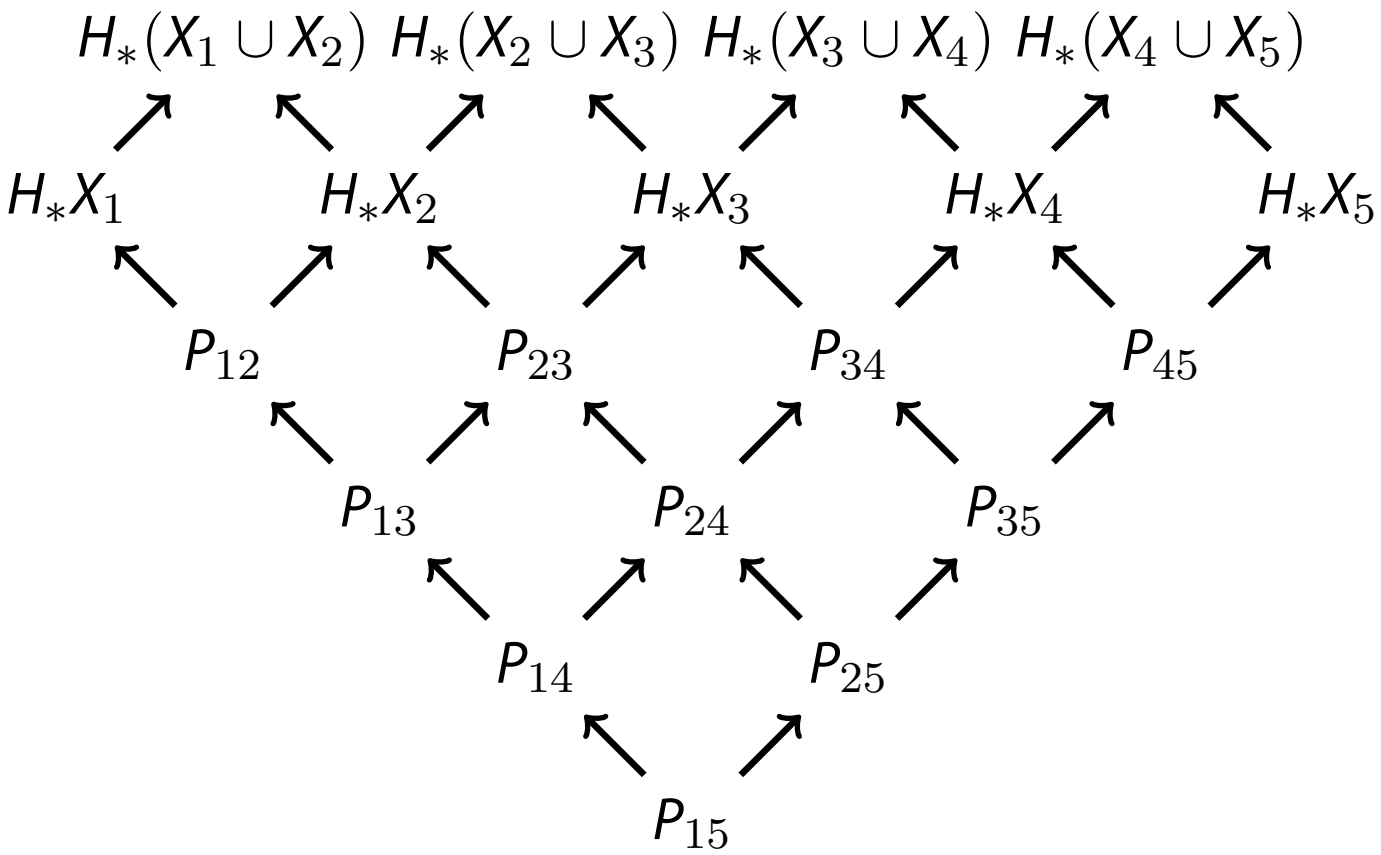
Algorithm overview



Algorithm overview



Algorithm overview



Algorithm overview

1. Compute all homology groups.
2. Compute all induced maps $H_*X_i \to H_*(X_i \cup X_j)$.
3. Compute pullbacks repeatedly until all layers processed.
4. Compute cokernels of the pullback maps.
   This picks out generators where they take effect.

Outline

Sample code

A serial implementation in GAP with comment annotations for parallelization with HPC-GAP exists.

Access with Mercurial: `hg://hg.gap-system.org/geometry`

Annotated code resides in `hpczz.g`

Shared but provably disjoint array write access. Shared array read access.

Code: Assumptions

The functionality is parametrized on:

- A sequence of pointclouds `pts`.
- A globally chosen Vietoris-Rips radius `eps`.
- A globally chosen top interesting dimension `d`.
- Some field `field`.

Homology setup

Computing homology is embarrassingly parallel; no dependencies between the spaces. `ParList` or dependency-free task launches. Needs to be done both for each single space, and each union of neighbouring spaces.

```
SpaceKernel := function(array, index, pts)
    local graph, complex;

    graph := VietorisRipsGraph(pts, eps);
    cpx := VietorisRipsIncremental(graph, d);
    array[index] := rec( cc := CreateChainComplex(cpx, field),
                         spxs := cpx,
                         graph := graph );
end;
```

Inclusion maps setup

This step is specific to the union zigzag; assumptions on vertex numbering come from how unions are coded. Step `index` depends on the `SpaceKernel` for `index` and `index+1`.

```
UnionKernel := function(array, index, spaces, unions)
    local uu;

    uu := CreateUnion(spaces[index].spxs,
                      spaces[index].graph.nV,
                      spaces[index+1].spxs,
                      unions[index].spxs,
                      field);
    array[index].right := uu[1];
    array[index+1].left := uu[2];
end;
```

Homology computation

Homology computation locally is still internally dependency-free; but relies on the corresponding `SpaceKernel` having finished.

```
HomologyKernel := function(array, index, spaces)
    local p, h, z;
    p := NaturalHomomorphismBySubspace(
                 Kernel(spaces[index].cc),
                 Image(spaces[index].cc));
    h := Image(x1p);
  z := List(Basis(x1h), v -> PreImagesRepresentative(x1p, v));
    array[index].space := h;
    array[index].cycles := z;
    array[index].proj := p;
end;
```

Induced maps

```
CreateMap := function(from,to,unionmap,proj)
  return LeftModuleHomomorphismByImages(
    from.space, to.space, Basis(from.space), List(List(from.cycles,
      v -> Image(unionmap, v)), w -> Image(proj, w)));
 end;
HomologyMapKernel := function(array, idx, state)
    local f, g, sl;
    sl := state.layers; f := fail; g := fail;
    if index < Length(array) then
        f := CreateMap(sl[2][idx], sl[1][idx],
                       state.unionmaps[idx][1], sl[1][idx].proj);
    fi;
    if index > 1 then
        g := CreateMap(sl[2][idx], sl[1][idx],
                     state.unionmaps[idx-1][2], sl[1][idx-1].proj);
    fi;
    array[index].left := g; array[index].right := f;
end;
```

Pullbacks

The function here is almost trivial; but the dependency structure more intricate.

```
PullbackKernel := function(state, depth, index)
    state.layers[depth][index] := Pullback(
        state.layers[depth-1][index].right,
        state.layers[depth-1][index+1].left);
end;
```

The `depth/index` invocation of the `PullbackKernel` depends on the `PullbackKernel` invocation for `depth-1/index` and `index+1`.

The first layer depends on the correspondingly indexed induced maps having been computed.

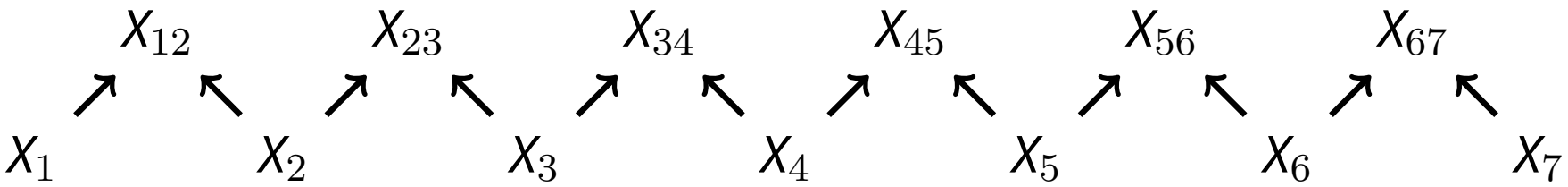
Cokernels

A final step, for a nice and complete presentation, is to compute cokernels, isolating the essential content of the computed barcode.

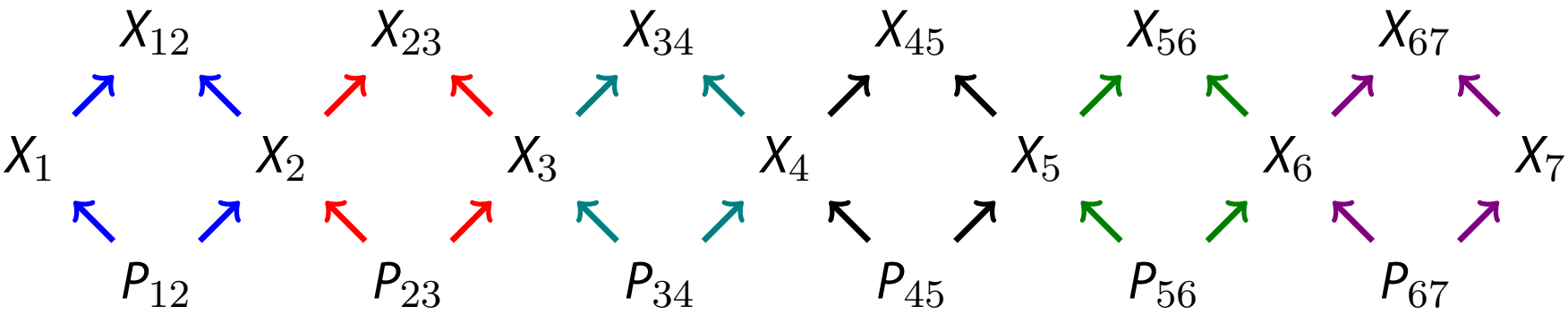
```
CokernelKernel := function(state, depth, index)
    local quot, hf, hh;

    quot := Subspace(state.layers[depth][index].space, []);
    if index <> Length(state.layers[depth]) then
        quot := quot + ImagesSet(state.layers[depth+1][index].left,
                        state.layers[depth+1][index].space);
    fi;
    if index <> 1 then
        quot := quot + ImagesSet(state.layers[depth+1][index-1].right,
                        state.layers[depth+1][index-1].space);
    fi;
    hf := NaturalHomomorphismBySubspace(
                  state.layers[depth][index].space,
                  quot);
    hh := Image(hf);
    state.zz[depth][index].hf := hf;
    state.zz[depth][index].hh := hh;
end;
```

Outline

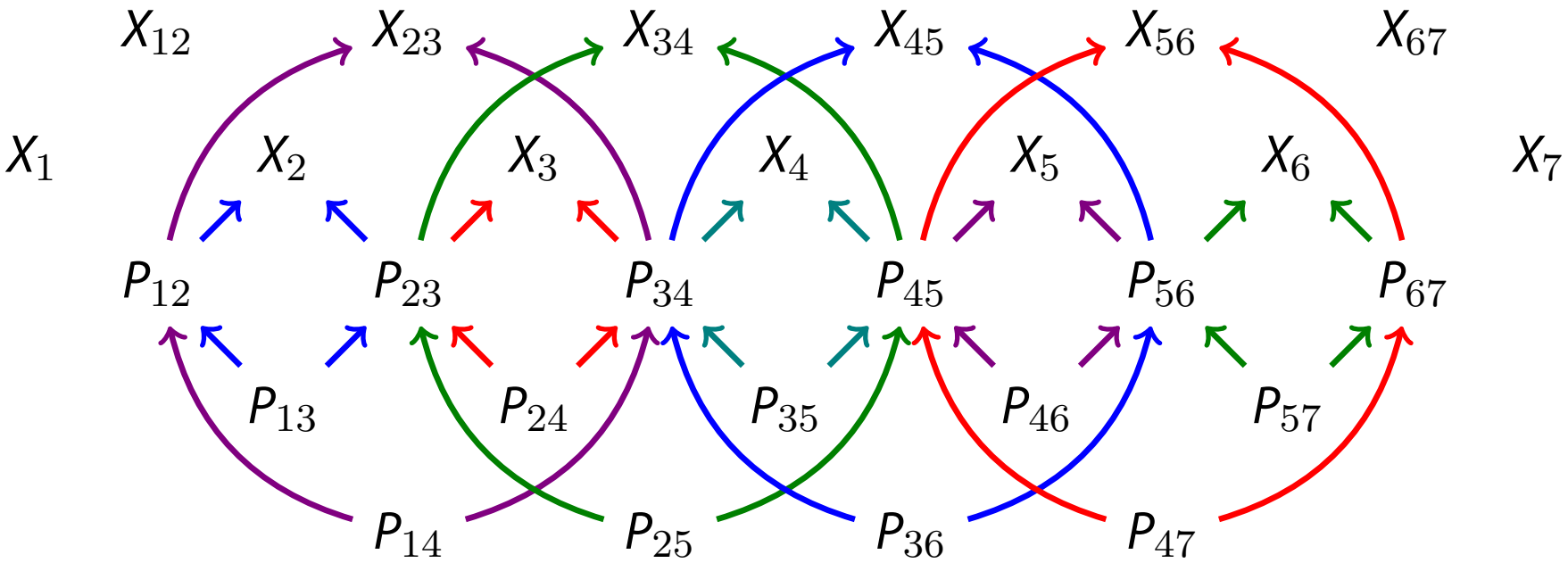
Logarithmic time; assuming quadratic resources



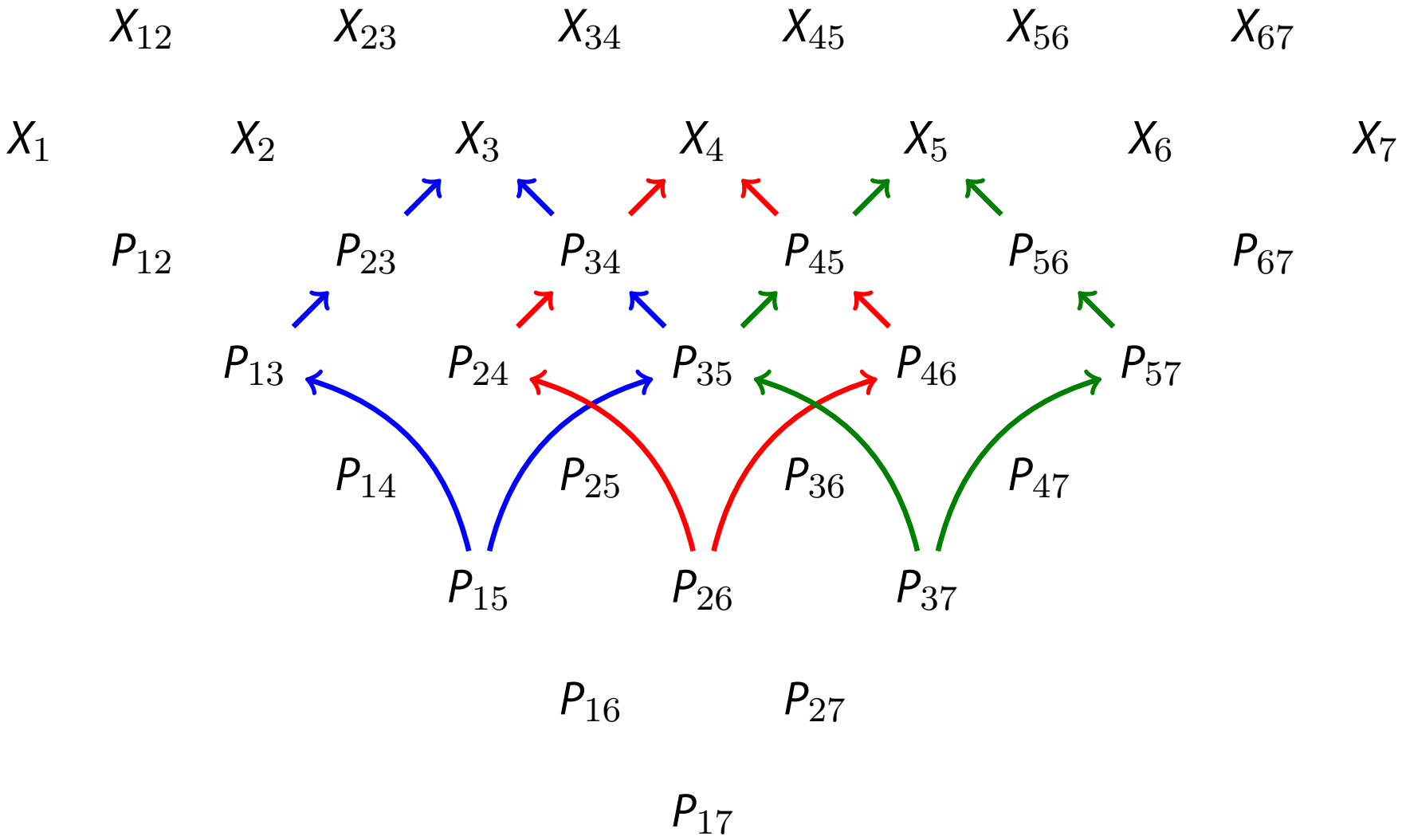
Logarithmic time; assuming quadratic resources



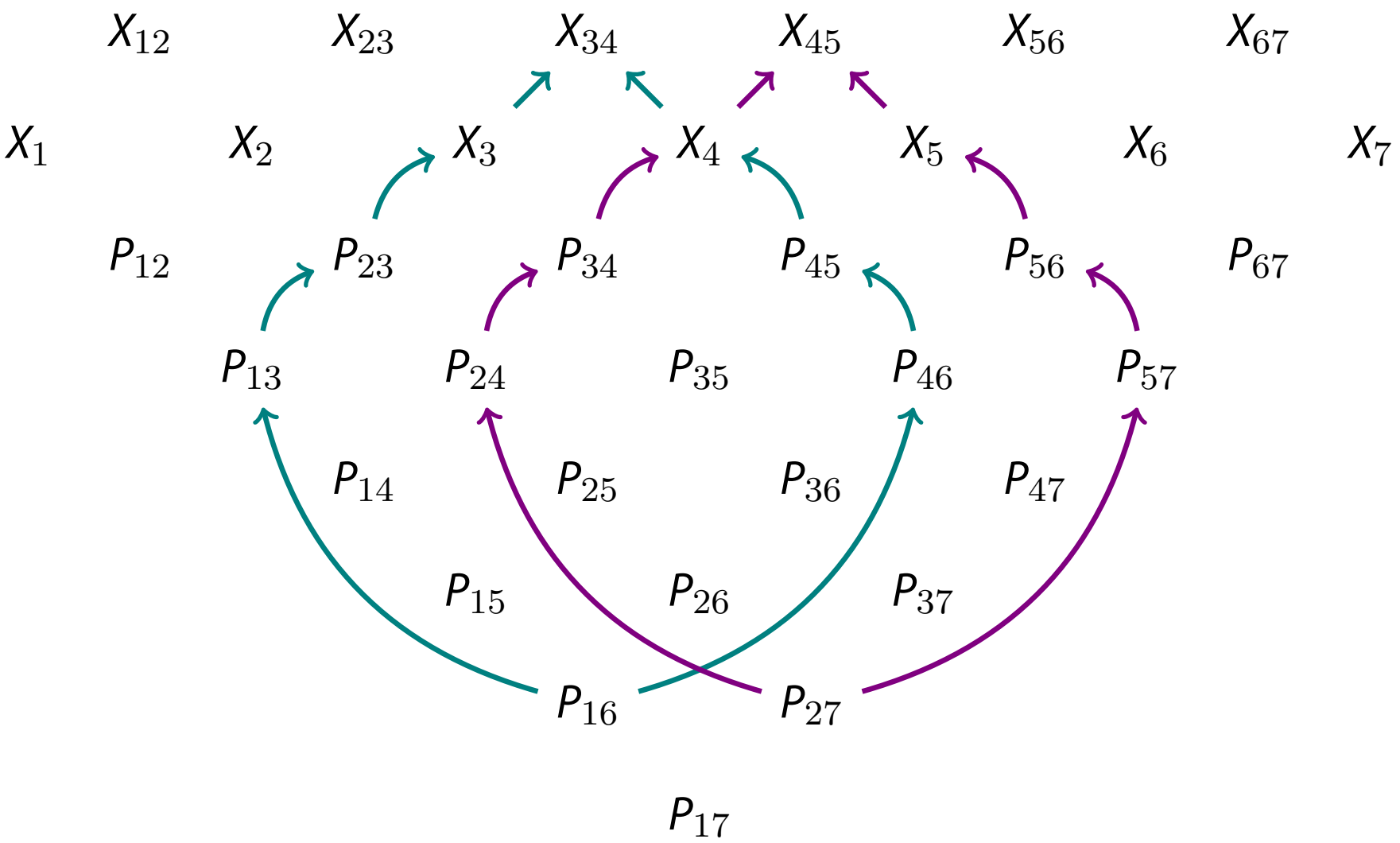
Logarithmic time; assuming quadratic resources



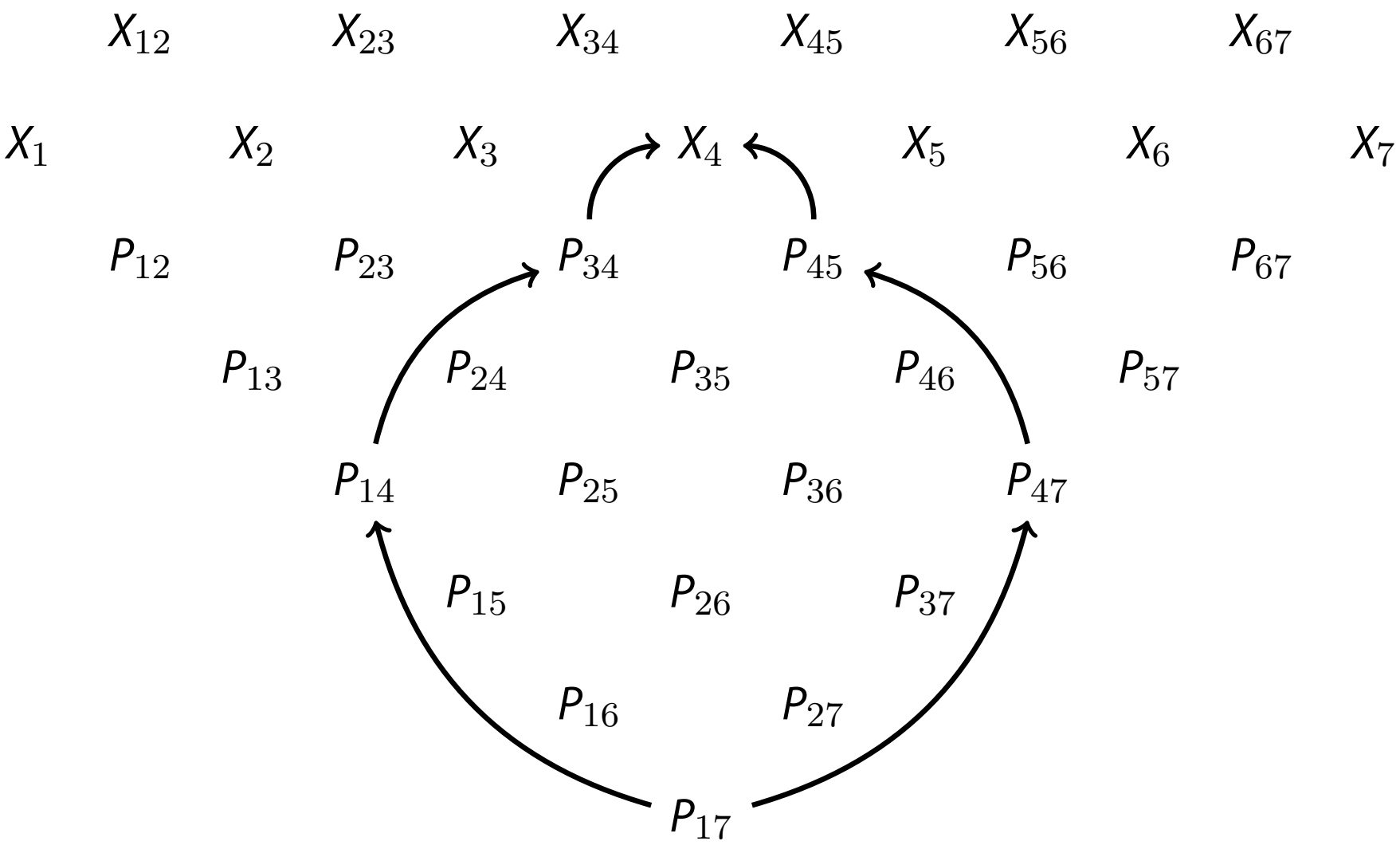
Logarithmic time; assuming quadratic resources



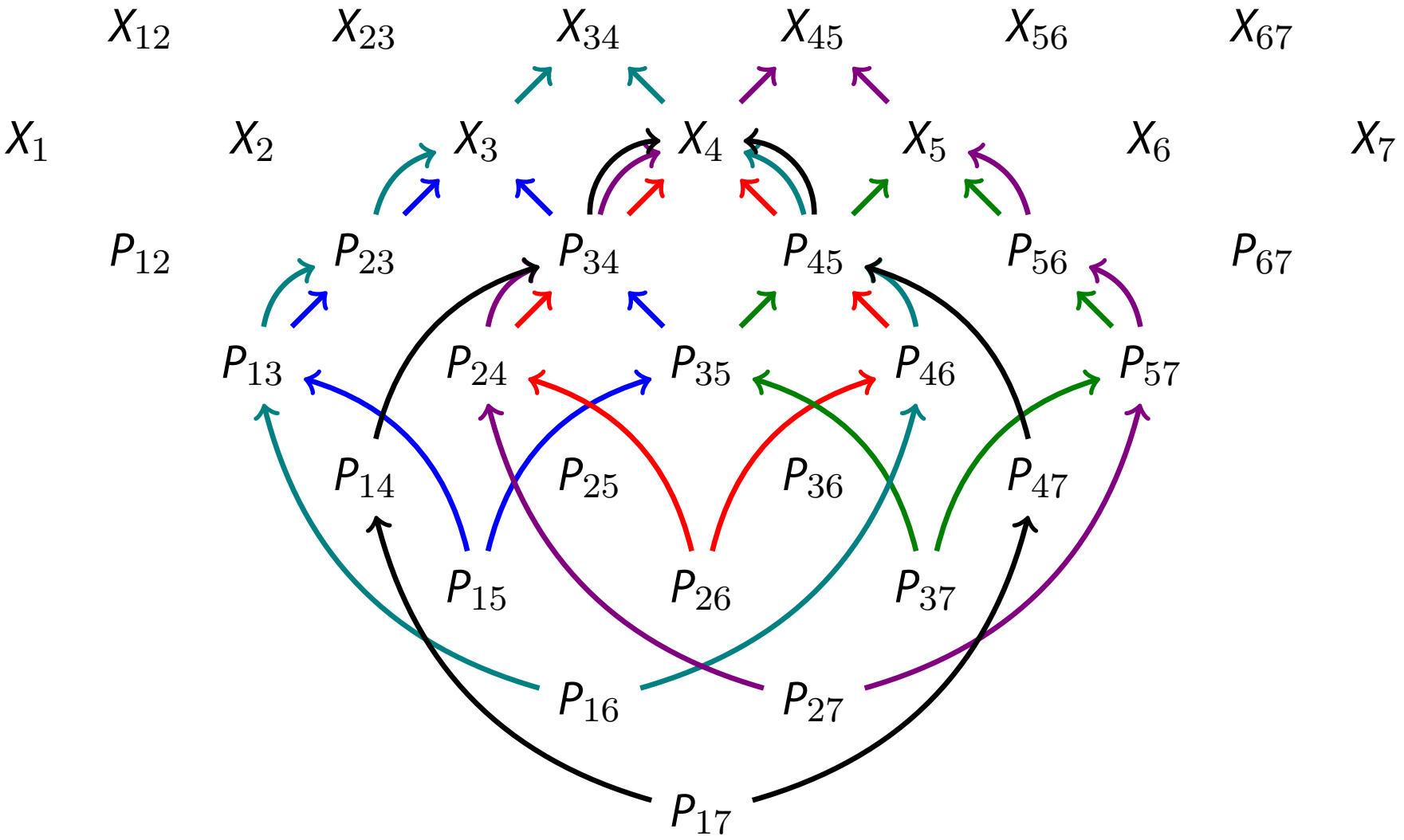
Logarithmic time; assuming quadratic resources



Logarithmic time; assuming quadratic resources



Logarithmic time; assuming quadratic resources



Complexity analysis

Assume n spaces; $n-1$ unions. Largest space has m simplices. Suppose that b_k is the upper bound on the number of classes persisting at least k steps.

Homology computation Time $O(m^\omega)$. Processors $2n-1$.

Maps computation Time $O(m^\omega)$. Processors $2n-2$.

Pullbacks, 1st round Time $O(b_1^\omega)$. Processors $n-1$.

Pullbacks, 2nd round Time $O(b_2^\omega + b_3^\omega)$. Processors $(n-2)+(n-3)$.

Pullbacks, kth round Time $O(\sum_{i=2^{k-1}}^{2^k-1} b_i^\omega)$. Processors $\sum_{i=2^{k-1}}^{2^k-1} n-i$.

The computation ends after $\lceil \log_2 n \rceil$ rounds.

ω is the exponent of matrix multiplication complexity.