A SUPPLEMENTARY MATERIALS

A.1 Derivation of Worse Case Insertion Performance

The number of combinations at each level l for a k-simplex is given by the binomial theorem, $\binom{k}{l}$. In the worst case, where the simplicial complex is empty, the total number of nodes created for inserting a k-simplex is $\sum_{l=0}^{k} \binom{k}{l} = 2^k$. The number of edges to represent all topological relations is then given by $\sum_{l=0}^{k} l\binom{k}{l} = k \cdot 2^{k-1}$.

A.2 Templated Insertion Algorithm

Pseudocode for the algorithms presented in this manuscript have been vastly simplified in order to facilitate understanding. For example Algorithm 1, while the non-templated version is appears straightforward, it is impossible to be implemented in C++ directly, due to several typing related issues. First, the function prototype for `insert()` requires the `rootSimplex` as the second argument. Simplices at different levels have different types and `insert()` must be overloaded. Similarly the variable `newSimplex` and function `createSimplex()` must know the type of simplex which will be created at compile time.

The actual implementation uses variadic templates to resolve the typing issues. As an example, templated pseudocode for simplex insertion (Algorithm 1) is shown in Algorithm 6. Not only does the templated code automatically build the correct overloaded functions, but it provides many optimizations.

The step-by-step insertion of tetrahedron {1,2,3,4} is shown in Figure 6. Numbered red lines correspond to `newNode` and `root` in function `insertNode()`. Skinny black lines are the topological relations inserted by `backfill()`.

ALGORITHM 6: Templated pseudocode implementation of Algorithm 1.

Input: $keys[n]$: Indices of n simplices to describe new simplex s,
$\mathcal{F}$: simplicial complex
Output: The new simplex s

```
/* User function to insert simplex {keys}                                  */
Function insert<n>(keys[n]){
    return setupForLoop<0, n>(root, keys) /* 'root' is the root node       */
}
// The following are private library functions...
/* Array slice operation. Algorithm 1: keys[0:i]                           */
Function setupForLoop<level, n>(root, keys){ /* General template           */
    return forLoop<level, n, n>(root, keys) /* Setup the recursive for loop */
}
Function setupForLoop<level, 0>(root, keys){ /* Terminal condition $n = 0$   */
    return root
}
/* Templated for loop. Algorithm 1: for (i = 0; i < n; i++)                */
Function forLoop<level, antistep, n>(root, keys){ /* For loop for antistep */
    /* $n - antistep$ defines next key to add to root                       */
    insertNode<level, n-antistep>(root, keys)
    return forLoop<level, antistep-1, n>(root, keys)
}
Function forLoop<level, 1, n>(root, keys){ /* Stop when antistep = 1       */
    return insertNode<level, n-1>(root, keys)
}
Function insertNode<level, n>(root, keys){ /* Insert a new node            */
    v = keys[n]
    if ($root \cup v \in \mathcal{F}$){
        newNode= root.up[v]
    }
    else{ /* Add simplex $root \cup v$                                      */
        newNode = createSimplex<n>() /* Create a new node, n-simplex newNode */
        newNode.down[v] = root /* Connect boundary relation                */
        root.up[v] = newNode /* Connect coboundary relation                */
        backfill (root, newNode, v)/* Backfill other topological relations */
    }
    /* Recurse to insert any cofaces of newNode. Algorithm 1: insert(keys[0:i],
       newSimplex)                                                          */
    return setupForLoop<level+1, n>(newNode, keys)
}
Function backfill<level>(root, newNode, value){ /* Backfilling pointers to other parents */
    for (currentNode in root.down){
        childNode = currentNode.up[value] /* Get simplex $currentNode \cup value$ */
        newNode.down[value] = child /* Connect boundary relation           */
        child.up[value] = newNode /* Connect coboundary relation           */
    }
}
```

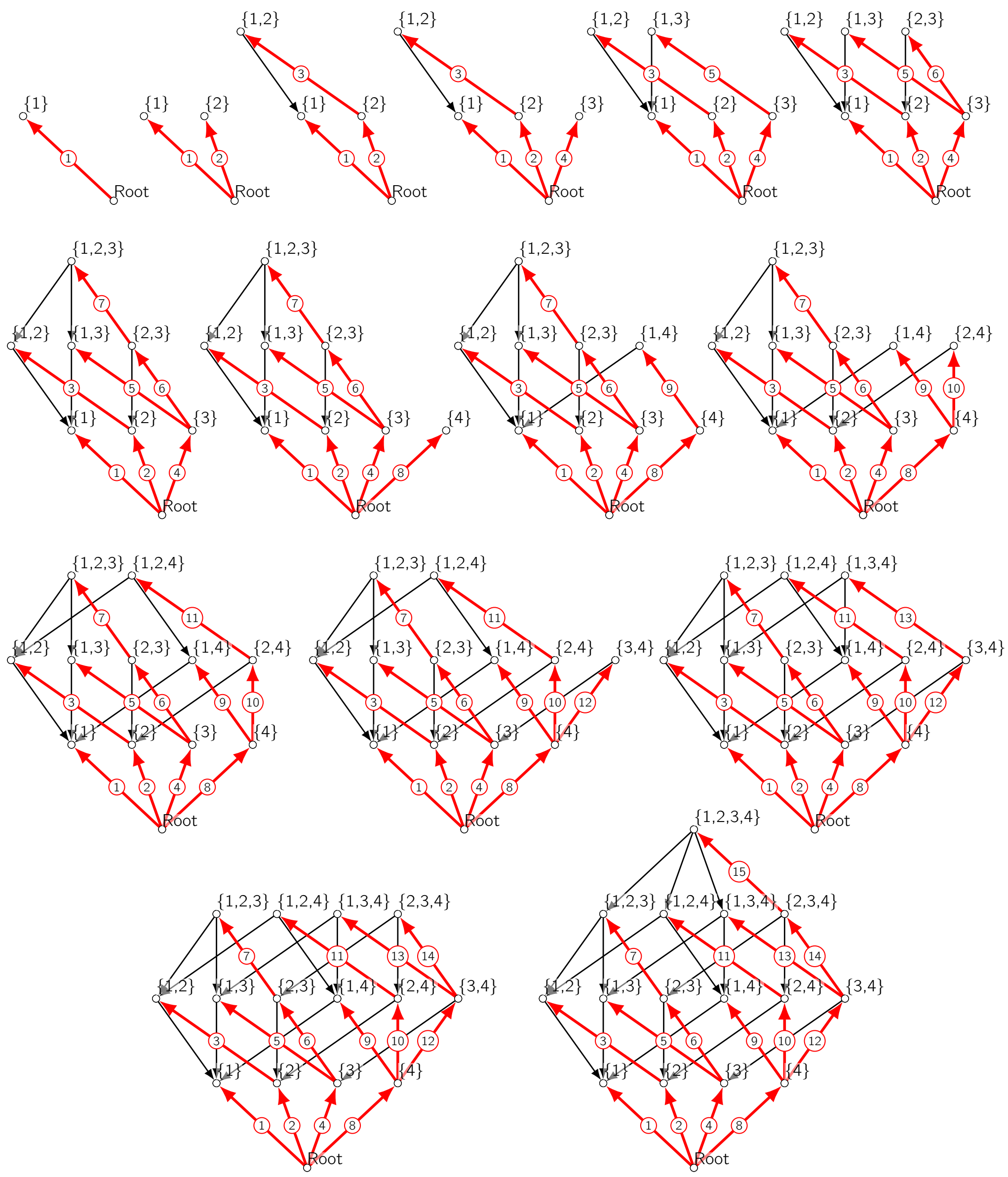

Fig. 6. The Hasse diagrams for the step-by-step insertion of tetrahedron {1,2,3,4} by Algorithm 1. Red lines represent the order of creation for each simplex. The skinny black lines represent where connections to parent simplices are backfilled.

A.3 Code for Getting Neighbors by Adjacency

The C++ code for collecting the set of k-simplices sharing a common coface with simplex s. A function call to `neighbors_up()` calls the following code which serves primarily to help the compiler deduce the dimension, k, of s.

```
template <class Complex, class SimplexID, class InsertIter>
void neighbors_up(Complex &F, SimplexID s, InsertIter iter)
{
    neighbors_up<Complex, SimplexID::level, InsertIter>(F, s, iter);
}
```

With the simplex dimension determined, we call an overloaded function which defines the operation for a k-simplex.

```
template <class Complex, std::size_t level, class InsertIter>
void neighbors_up(
        Complex &F,
        typename Complex::template SimplexID<level> s,
        InsertIter iter)
{
    for (auto a : F.get_cover(s))
    {
        auto id = F.get_simplex_up(s, a);
        for (auto b : F.get_name(id))
        {
            auto nbor = F.get_simplex_down(id, b);
            if (nbor != s)
            {
                *iter++ = nbor;
            }
        }
    }
}
```

Neighbors of s are pushed into an insert iterator provided by the user. In this fashion, depending upon the container type the iterator corresponds to, the user can specify whether or not duplicate simplices are returned (`std::vector`) or not (`std::set`).