Hosoya Index

Problem Statement

The Hosoya index $Z$ is the number of sets of nonadjacent bonds in a molecule [http://en.wikipedia.org/wiki/Hosoya_index]. If we model a molecule as a graph, then the atoms of the molecule are the vertices ($V$) and the adjacent bonds between atoms are the edges ($E$) of the complete graph. In other terms, the Hosoya index of a graph $Z(G(V,E))$ is the total number of matchings within the graph, where a matching is a subset of edges that do not share a vertex.

Computation of the Hosoya index of a given graph can be done by summing up the number of edge subsets contained within all the $k$-matchings, where a $k$-matching is the set of subsets of exactly $k$ ($0 \leq k \leq |E|$) edges.

Problem: Write a threaded application to input a graph from a file and compute the Hosoya index of the input graph. Input graphs will be connected and undirected.

The Hosoya index may be computed in an amount of time exponential in the number of edges of the graph. A naïve algorithm for calculation may look as:

```c
int calculate_hosoya_index(G)
{
    int result = 0;
    for each possible combination X of edges in G
    {
        if (is_a_valid_matching(X, G))
            result += 1;
    }
    return result;
}
```

Implementation

The Baseline Algorithm

First of all, the naïve algorithm is a way too naïve. The main drawback is that it enumerates all possible combinations of edges, thus it's computation complexity is somewhere near $O((N^2)*(2^N))$.

The idea is to build a goal-seeking exhaustion algorithm that enumerates only valid matchings. The baseline algorithm that I use is as follows (it uses edge adjacency list representation, http://en.wikipedia.org/wiki/Adjacency_list):

```c
int calculate_hosoya_index(vector<vector<int>> const& adjacency_list)
{
    // all edges are initially available for inclusion
    vector<int> available_edges (adjacency_list.size());
    for (size_t i = 0; i != available_edges.size(); i += 1)
        available_edges[i] = i;
    // "+1" is for the empty set of edges
    return 1 + calculate_hosoya_index_recursive(adjacency_list, available_edges);
}

int calculate_hosoya_index_recursive(vector<vector<int>> const& adjacency_list, vector<int> const& available_edges)
```
```cpp
int result = 0;
for (size_t i = 0; i != available_edges.size(); i += 1)
{
    // count inclusion of the i-th available edge
    result += 1;
    vector<int> const& adjacency = adjacency_list[available_edges[i]];

    // calculate available edges after inclusion of the edge
    vector<int> next_available_edges;
    set_difference(available_edges.begin(), available_edges.begin(),
                   adjacency.begin(), adjacency.end(),
                   back_inserter(next_available_edges));

    // if the set of available edges is not empty,
    // apply the procedure recursively
    if (next_available_edges.size())
        result += calculate_hosoya_index_recursive(adjacency_list,
                                                    next_available_edges);
}
return result;
```

The computational complexity of the algorithm is \(O(E*Z)\) (where \(Z\) is the Hosoya index, and \(E\) is the number of edges), i.e. for each valid matching we execute the set_difference() function that takes \(O(E)\) time.

It's worth noting that the algorithm is easily parallelizable, because it's possible to divide the exhaustion into completely independent pieces. Namely, (1) enumerate all valid matchings with edge 0 included, and (2) enumerate all valid matchings with edge 0 excluded; then each piece can be further divided based on edge 1, and so on. The resulting division forms well-balanced tree of independent work pieces that can be executed in parallel.

**Still Too Slow**

The baseline algorithm is somehow faster than the naïve algorithm, but still has exponential complexity, thus takes basically infinite amount of time on large inputs. What we want is a more significant optimization.

If a graph \(G\) consists of two independent components \(G_1\) and \(G_2\), i.e. edges in \(G_1\) are nonadjacent to edges in \(G_2\). Then matchings in \(G_1\) and in \(G_2\) can be enumerated independently, thus the Hosoya index \(Z(G)\) is a product of \(Z(G_1)\) and \(Z(G_2)\). For example, if \(G_1\) contains 50 edges, and \(G_2\) contains 50 edges, computational complexity of direct computation of \(Z(G)\) is \(O(2^{100})\). While complexity of computation that takes into account independence of \(G_1\) and \(G_2\) is \(O(2 * 2^{50}) = O(2^{51})\), that saves us maybe several thousands of years or so.

Ok, that's nice, but we have a connected graph.

The idea is to do the sparsest cut of an input graph [http://en.wikipedia.org/wiki/Cut_%28graph_theory%29](http://en.wikipedia.org/wiki/Cut_%28graph_theory%29). The sparsest cut divides edges of the graph into 3 sets: edges that are a part of the cut, and 2 mutually independent sets. Let's illustrate this with a picture:
Red edges form the cut. Blue and green edges form 2 mutually independent sets, i.e. no blue edge is adjacent to a green edge. Let's call red edges the M (middle) set, blue edges the L (left) set, and green edges the R (right) set.

Then, we enumerate all possible matchings for the set M. For each such matching X we determine subset L0 of all edges in L that are nonadjacent to X. And similarly the subset R0 of all edges in R that are nonadjacent to X. Then we calculate the Hosoya index Z for L0 and for R0 independently, multiply Z(L0) by Z(R0) and add to the result.

Note that this optimization can be applied recursively when we calculate Z(L0) and Z(R0).

Here is a pseudo-code implementation of the idea:

```c
int calculate_hosoya_index(G)
{
    <L,M,R> = do_sparsest_cut(G);
    result = 0;
    for each matching X of edges in M
    {
        L0 = determine all edges in L that are nonadjacent to X
        res1 = calculate_hosoya_index(L0);

        LR = determine all edges in R that are nonadjacent to X
        res2 = calculate_hosoya_index(R0);

        result += res1 * res2;
    }
    return result;
}
```

The computational completely of the algorithm is \( C(E) = O(2^M \times (C(L) + C(R))) \) (E is the total number of edges; L, R and M mean sizes of the sets). It's worth noting that M can be somewhere between 1 and E/2. Sparse graphs tend to produce low values of M; if on each recursion step we get M of low constant size, then computational complexity becomes basically \( O(E) \). While for fully connected graphs \( M = E/2 \), and computational complexity becomes \( O(2^{2E/3}) \).
Parallelization

It's possible to apply different parallelization strategies to the algorithm, or even combine several of them. For example, it's possible to organize a pipeline processing, where first serial stage produces matchings of edges in M, and then second parallel stage calculates $Z(L0)$ and $Z(R0)$ for each matching. Or it's possible to calculate $Z(L0)$ and $Z(R0)$ independently in parallel.

However, a rule of thumb says that parallelization must be applied on the highest possible level and provide maximum independence between tasks. So I've chosen the following parallelization strategy. I divide all matchings of edges in M into two roughly equal independent pieces (all matchings that include edge 0, and all matchings that do not include edge 0). Then, divide each of these pieces into two roughly equal independent pieces again (based on edge 1). And so on, until size of a piece reaches some predefined threshold. Each such piece can be executed in parallel. Here is an illustration:

![Parallelization Diagram]

Such recursive decomposition plays very nicely with all parallelism support libraries based on Cilk-style [http://supertech.csail.mit.edu/cilk/] work-stealing schedulers, for example, Intel TBB, Microsoft PPL, Microsoft TPL, Cilk++, Java Fork/Join, etc. I've chosen Intel TBB as parallelism support library. In particular I use structured_task_group component and an ability to use C++0x lambda functions as tasks, that allowed me to parallelize the program by adding only 4 lines of code:

```cpp
int recursion(vector<vector<int>> const& adjacency_list_middle,
              vector<vector<int>> const& adjacency_list_left,
              vector<vector<int>> const& adjacency_list_right,
              vector<vector<int>> const& adjacency_list_middle_for_left,
              vector<vector<int>> const& adjacency_list_middle_for_right,
              vector<int> const& available_middle,
              vector<int> const& available_left,
              vector<int> const& available_right)
{
    // decide as to whether split or not
    if (available_middle.size() >= split_threshold)
    {
        // create a task group
        tbb::structured_task_group group;

        count_t result1 = 0;
        count_t result2 = 0;

        // spawn two tasks
```


```cpp
  group.run(tbb::make_task([&]() {
    // the first task contains matchings with the first available edge
    vector<int> next_available_middle (available_middle.begin() + 1, available_middle.end());
    
    // call the same function recursively
    result1 = recursion(adjacency_list_middle, adjacency_list_left,
                        adjacency_list_right,
                        adjacency_list_middle_for_left, adjacency_list_middle_for_right,
                        next_available_middle, available_left, available_right);
  }));

  group.run(tbb::make_task([&]() {
    // the second task contains matchings with the first available edge
    
    // since we included the first available edge
    // we exclude edges adjacent with it from available sets (middle, left and right)
    vector<int> next_available_middle;
    vector<int> const& adjacency_middle = adjacency_list_middle[available_middle[0]];
    set_difference(available_middle.begin(), available_middle.end(),
                   adjacency_middle.begin(), adjacency_middle.end(),
                   back_inserter(next_available_middle));

    vector<int> next_available_left;
    vector<int> const& adjacency_left = adjacency_list_middle_for_left[available_middle[0]];
    set_difference(available_left.begin(), available_left.end(),
                   adjacency_left.begin(), adjacency_left.end(),
                   back_inserter(next_available_left));

    vector<int> next_available_right;
    vector<int> const& adjacency_right = adjacency_list_middle_for_right[available_middle[0]];
    set_difference(available_right.begin(), available_right.end(),
                   adjacency_right.begin(), adjacency_right.end(),
                   back_inserter(next_available_right));
    
    // call the same function recursively
    result2 = recursion(adjacency_list_middle, adjacency_list_left,
                        adjacency_list_right,
                        adjacency_list_middle_for_left, adjacency_list_middle_for_right,
                        next_available_middle, next_available_left, next_available_right);
  }));

  // wait for both tasks to complete
  group.wait();

  // sum up the partial results
  return result1 + result2;
} else {

```
Performance

As expected, run time of the program directly depends on the size of the first (top-level) sparsest cut. In particular, if $M$ is the size of the first sparsest cut, then computation complexity is $O(2^M)$. For very sparse graphs $M$ is a small constant (1, 2, 3, ...) and computation completely degenerates to polynomial. For example, for a minimal spanning tree with 400 vertices (399 edges) run time of the program on my Intel Core2Duo P8500 2.5GHz is:

1 thread: 248 ms
2 threads: 144 ms

It's impossible to test the program on larger minimal spanning trees, because the result just does not fit into 64-bit integer.

For fully connected graphs, the size of the first sparsest cut $M = E/2$ ($E$ – number of edges). So it's expected that run time will quickly degrade for such graphs. Run time for fully connected graph with 18 vertices (153 edges) is:

1 thread: 5943 ms
2 threads: 3111 ms

Run time for fully connected graph with 19 vertices (171 edges) is:

1 thread: 24047 ms
2 threads: 12506 ms

Fortunately, for real chemical molecules (for which Hosoya index is expected to be calculated) size of the sparsest cut is expected to be small, because of the sparseness and locality (each atom is connected to 1-7 local neighbors). So as a final test I've got a reasonably dense graph with 24 vertices and 140 edges and ran it on Intel Parallel Universe Portal (the service allows everyone to test a program on an 2-processor/8-core/16-thread machine, which is reasonably enough for scalability testing) [http://paralleluniverse.intel.com/Home]. Here are the results of the run:
So the final speedup is 9.2. Taking into account that the machine contains 8 physical cores, I consider parallelization as a successful.