Algorithms used by CSM

Itay Zandbank

August 26, 2015

Abstract

This document describes various algorithms used by the CSM program. It will help anybody looking at the CSM source code figure out what's going on. The document describes the Python version of CSM, not the older C++ version. There's no explanation of the mathematics or chemistry behind the algorithms.

1 Atom Equivalency Class Separation

The CSM program enumerates over permutations during calculation. These are atom permutations of atoms in the molecule. Instead of going over all $n!$ permutations, CSM divides the atoms into groups of interchangable atoms, going over permutations of each group, instead of the entire $n!$ space.

The separation is done by atom type (if atom types are considred) and the *Connectivity* of each atom. Connectivity is defined recursively two atoms are equal if the are of the same type, have the same number of neighbors, and all their neighbors are equal. So if A is connected to B and C, and D is connected to E and F, A and B are equal iff B is equal to E or F, and C is equal to F or E. If D is connected to E, F and G, it is not equal to A.

CSM implements an iterative algorithm to find these groups. The algorithm is implemented in the Molecule class. It is a dynamic programming algorithm. Each iteration improves on the equivalency classes created in the previous iteration.

We define two atoms to be $\mathbf{similar}_i$ if they are found to be similar in the i^{th} iteration. Atoms that are equivalent if they are similar after the last iteration of the algorithm.

The first iteration builds the first grouping of atoms as follows: $atom_i\ \mathbf{similar}_0\ atom_j$ if they have the same symbol and the same number of neighbors.

In further iterations, $atom_i\ \mathbf{similar}_n\ atom_j$ if each of $atom_i$'s neighbors has a $\mathbf{similar}_{n-1}$ atom in one of $atom_j$'s neighbors

The algorithm builds these similarities groups iteratively. The similary groups are stored in the `groups` list. `groups[0]` is the first similary group, `groups[1]` is the second and so forth. Each group is a list of atom indices. So, for example, `groups` can be `[[0,1],[2,3,4]]`, meaning we have 5 atoms in the molecule, the first two form a similarity group, as do the last three.

The algorithm first builds the groups of $\mathbf{similar}_0$ atoms. Then, each iteration n looks like this:

- Go over all groups.
- In each group, compare the first atom to all other atoms
- All atoms that are not $\mathbf{similar}_{n-1}$ to the first atom are split to a new group.

The algorithm stops once no new group has been created.

2 Permutation Enumeration

Once we have the equivalency classes calculated, we have a much smaller permutation space to enumerate. Instead of enumerating over $n!$ iterations, we can enumerate over $n_0! * n_1! * ...$ permutations, each n_i being the size if the i^{th} equivalency class.

The space can be further reduced because there are limitations on the cyclic structure of the permutation. Permutations are only allowed to have cycles of specific lengths - namely - the symmetry operation order and (under some circumstances) - 2.

CSM builds such permutations for each equivalency class, and then combines them to construct permutations for the entire molecule. This combination is in fact a cartesian product of the permutations of each equivalency class.

2.1 Permutations for one equivalency class

The implementors of the C++ version of CSM went through a lot of trouble to enumerate through ell permutations. They did not want to store all permutations in memory and instead wanted to generate them in-place. This forced them to use a barely decipherable non-recursive algorithm. Thankfully, Python 3 has the `yield from` construct, which makes building recursive generators very straightforward. This allowed us to use much simpler recursive algorithms without requiring any more memory.

2.1.1 Cycle Structures

We define a *Cycle Structure* - the break down of the permutation into cycles and stationary points. For example, the cycle structure `[[0, 1],[2, 3],[4]]` is a permutation with two cycles (elements 0 and 1 change palces, and elements 2 and 3 change places) and one stationary point (element 4). The cycle structure `[[0],[1],[2],[3],[4]]` describes a permutation with 5 stationary points, namely the identity permutation `(0,1,2,3,4)` .

Given a permutation size n and allowed cycle sizes, we first generate all possible cycle structures. This is done recursively:

- Choose the cycle of the first element (cycle size and other elements in the cycle)
- Build a permutation of the elements not in the cycle, and combine.

For permutations of size 5, with cycles of size 3 (and stationary points), the possible cycle structures are:

1. `[[0], [1], [2], [3], [4]]`
2. `[[0], [1], [2, 3, 4]]`
3. `[[0], [1, 2, 3], [4]]`
4. `[[0], [1, 2, 4], [3]]`
5. `[[0], [1, 3, 4], [2]]`
6. `[[0, 1, 2], [3], [4]]`
7. `[[0, 1, 3], [2], [4]]`
8. `[[0, 1, 4], [2], [3]]`
9. `[[0, 2, 3], [1], [4]]`
10. `[[0, 2, 4], [1], [3]]`
11. `[[0, 3, 4], [1], [2]]`

We need to enumerate over permutations with a specific cycle structures. Again we break the problem down into smaller problems and solve the problem recursively. We enumerate over each cycle, and combine all of them.

2.1.2 Cyclical Permutations - Circles

We need to enumerate over a cycle. A cycle of size k can be seen as a permutation of size k that consists of one cycle and has no stationary points. Such permutations are sometimes called *circles* and sometimes even *necklaces* because they can be viewed as a necklace:

$$p_0 \leftarrow p_1 \leftarrow p_2 \leftarrow ...p_{n-1} \leftarrow p_0$$

. Each element is shifted to the element before it in the necklace, and the first element is moved to the last.

Since each circle can be expressed as a necklace above, there are $(n-1)!$ circles of size n - we can choose any starting point so we fix it to be 0.

Building a circle permutation of size n is easy. We go over all the necklaces (by enumerating over all permutations of size $n-1$ and fixing the first element to be 0), then we turn each necklace into a permutation.

For example, the necklace $0 \leftarrow 4 \leftarrow 2 \leftarrow 3 \leftarrow 1 \leftarrow 0$ yields the permutation `(1, 3, 4, 2, 0)`

Here are all the circles of size 4:

1. `(1, 2, 3, 0)`
2. `(1, 3, 0, 2)`
3. `(2, 3, 1, 0)`
4. `(2, 0, 3, 1)`

5. `(3, 2, 0, 1)`

6. `(3, 0, 1, 2)`

Note that these are *all* permutations of size 4 that have a cycle size of 4. All other permutations of size 4 have smaller cycles.

2.1.3 Combining circles into permutations

Combining circles into permutations is easy. We enumerate over all the circles of the first cycle, and then recursively combine the rest of the circles.

2.2 Combining Permutations of Equivalency Classes

This, too, is very easy to do recursively. We enumerate over all permutations of the first class, and then recursively combine the permutations of the rest of the classes.