Expanding Neighborhood Method

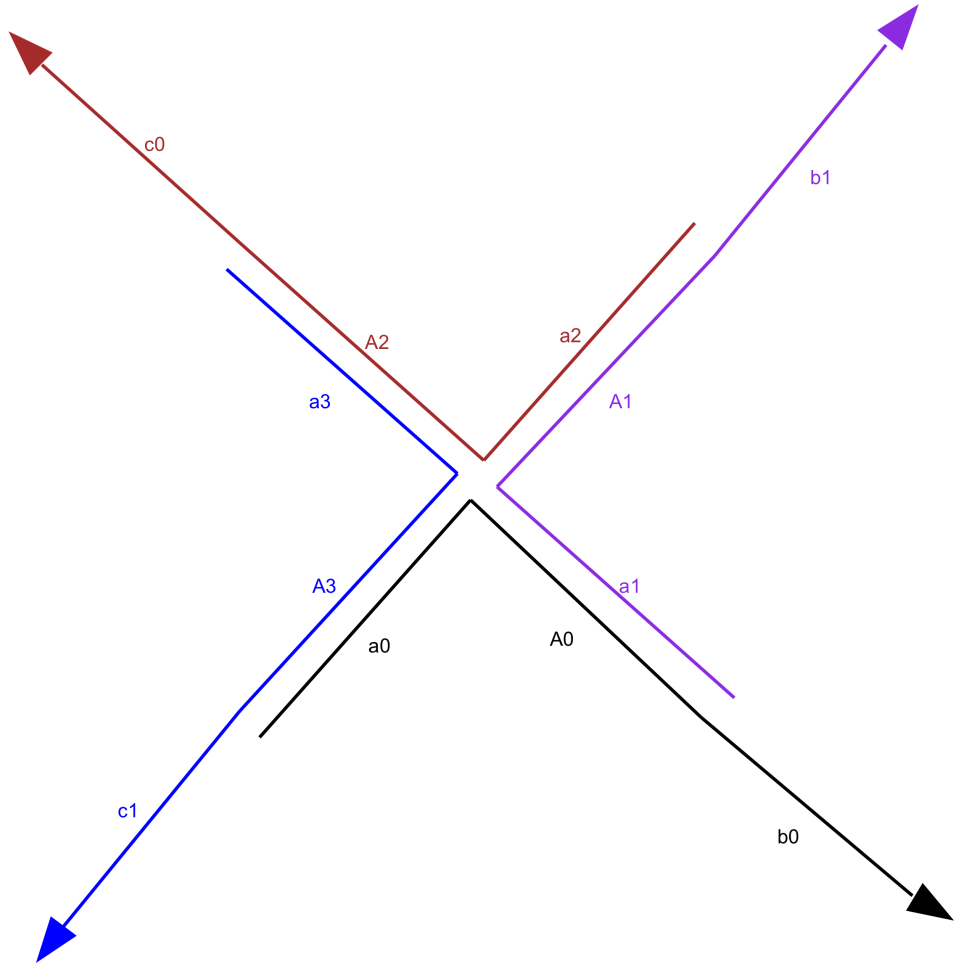
An improved algorithm for creating unique Component representations

In this report I describe the new algorithm developed for creating a unique representation of a Component. In tests with the old screening-algorithm, the process would take approximately 75% of execution time when performing an A* run on the "cube" component.

The old screening-algorithm is described in section 2.2.1 in my thesis report, and I will not go into any details of it here. The new algorithm however, takes on the same initial approach of uniquely selecting one segment on the component, and then using this segments unique component-iterator to create the uniquely ordered list of all segments in the component.

The new algorithm

- First list all segments in the Component.
- With every segment also hold its component-breadth-first iterator, and the iterators "next" element.
- Start a loop which terminates when there is only one segment element left in this list, or when the iterators have run out of elements. Iteration also stops if all remaining "next" segments compare equal on a local level, and the set of all segments covered by all remaining iterators equal the set of all segments in the component.
    - For every loop iteration, get the "next" segment of the segment's iterator.
    - Compare all "next" segments by their local neighborhood. That is, their own label, the label of any annealed segment, and any ligated segments.
    - For every unique local "next" segment's representation, select the least common representation.
    - If there are more than one least common representation, then select the one that compares lowest.
    - Remove all segment entries in the list with a "next" segment that does not have this local representation.
- If there is more than one segment element left in the list, select any one of these segments.



Old screening-algorithm example

In both of the following examples the component used is: `a0_A3,A0_a1,b0;a1_A0,A1_a2,b1;a2_A1,A2_a3,c0;a3_A2,A3_a0,c1` shown in the picture above.

In the old screening-algorithm, sub-sets of segments would be created for the different segment properties that exist. E.g. there would be a sub-set for all segments with label “a”, label “b”, label “c”, all annealed segments, all non-annealed segments, and so on. More sub-sub-sets would be creating by selecting more properties and taking the intersection of these sub-sets. E.g. a sub-set for all annealed segments with label “a”.

In the example above there will be two segment sets of size 4, namely the set of all segments labeled “a”, and the set of all segments labeled “A”. There are also two segment sets of size 2, namely the set of all segments labeled “b”, and all segments labeled “c”.

A ranking method would then decide which one of the sets of segments “b” or “c” would be discarded. The two remaining segments labeled “b” would then be placed in a list, the list would be sorted by the global segment comparison method, and the first segment in the sorted list would be selected.

April 21, 2010

New expanding-neighborhood-algorithm example

In the new algorithm, all segments are placed in a list with its breadth-first-iterator and the iterator's "next" element. The first "next" segment of all iterators is the starting segment. These segments are then compared locally. This means we have 5 different local representations. Using the legend segment:label(annealed:label,five:label,three:label) the representations are.

- a(A,None,A) – 4 of these
- A(a,a,b) – 2 of these
- A(a,a,c) – 2 of these
- b(None,A,None) – 2 of these
- c(None,A,None) – 2 of these

The least common local representation have 2 segments, this means a(A,None,A) is discarded. There are 4 different local representations with only two segments. These representations are sorted, and which ever compares lowest is selected. In the example A(a,a,b) is selected and all other segments are discarded. At the end of the first loop iteration we now only have 2 segments remaining.

The "next" segment in the A(a,a,b) segments iterator is the annealed "a" with local representation a(A,None,A) for both remaining segment iterators. The next "next" segment is the "a" ligated at the 5' end a(A,None,A) which again are equal for both iterators. The next "next" segment is the b(None,A,None).

The next "next" segment is the first neighbor's neighbor that is examined. In this case it is the annealed "a"-s ligated "A". This is when we finally have a difference. One of the iterators will find an A(a,a,b) and the other will return an A(a,a,c). These local "next" segment representations are sorted and the one that compares lowest is selected, in this case A(a,a,b).

The starting segment associated with the remaining (segment, iterator, next) tuple, is then selected.

Complexity Analysis

In the worst-case scenario for the expanding-neighborhood-algorithm all n iterators, iterate all n times all the way to the end, sorting n elements every time. At every iteration the local representation for all "next" segments are compared, i.e. a list is sorted. This gives us an over-all time-complexity of $O(n^2 log(n))$. See later comment on how to avoid this worst-case scenario, by using a special early termination criteria.

For the old screening-algorithm all segments would still remain after the screening, leaving all segments to be sorted. With n segments in the list $nlog(n)$ comparisons have to be done. The worst-case scenario for comparing two segments happens when they are identical. This means that both of their breadth-first-iterators have to all the way to the end, i.e. iterate over n elements. This adds another n to the over-all time-complexity, resulting in $O(n^2 log(n))$.

Results, Conclusion, Discussion

In the typical "cube" scenario, the execution time for unique component representation creation, has been pushed down from 75% to 35%.

With this method very little preparation needs to be done. All that is needed is creating a list O(n) and obtaining the segments breadth-first iterators O(1).

Already after the first iteration, most segments are thrown out with most components, since most segments compare differently in their closest neighborhood.

Note that no sorting has to be done. The way the segment sorting method works is in fact very similar to what is going on in this algorithm. When two segments are compared, their breadth-first iterators are used to step though the entire component. Here, iterators for all segments are run in parallel to minimize the number of local segment comparisons.

The reason it doesn't matter which segment is used if more than one remain when looping is finished, is because these remaining segments have compared equal through the entire component and are thus identical.

Iteration also stops if all remaining "next" segments compare equal on a local level, and the set of all segments covered by all remaining iterators equal the set of all segments in the component. This can be done because it can be concluded that there is symmetry between all remaining segment-elements in the list. If all segments have already been covered, then all "next" segments that remain to be iterated over in the individual iterators, will be segments that have already been compared. This termination criteria is useful for scenarios when many segments in a component compare equal, e.g. in a loop component with segments of equal labels. This added termination criteria improves time complexity in such a scenario from $O(n^2 log(n))$ where all n iterators have to do all n iterations where n segments are sorted, to $O(nlog(n))$ where sorting only has to be done once as after one iteration it can be concluded that all segments are identical.

Figure 1 shows plot of the worst-case scenario of both algorithms with the early termination. They both seem to have similar complexities, though the new algorithm is notably faster.

Figure 2 shows the plot of the worst-case scenario of both $O(nlog(n))$ algorithms without early termination. It can be seen that they have identical time-complexities and that the theoretical $O(nlog(n))$ complexity seems to be reasonable.

Figure 3 shows the typical components created during construction of a cube. It is clear that the new algorithm greatly improves performance.

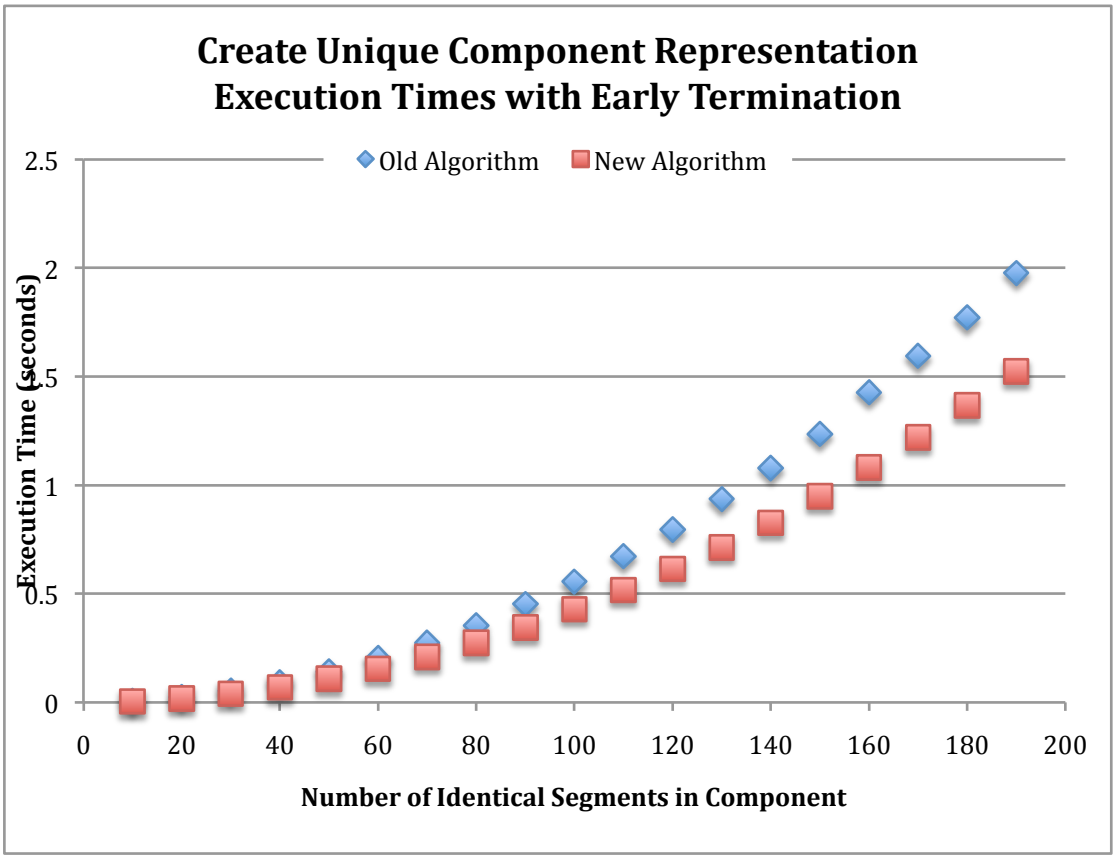


Figure 1. Creating unique component representations for components with identical segments, using early termination.

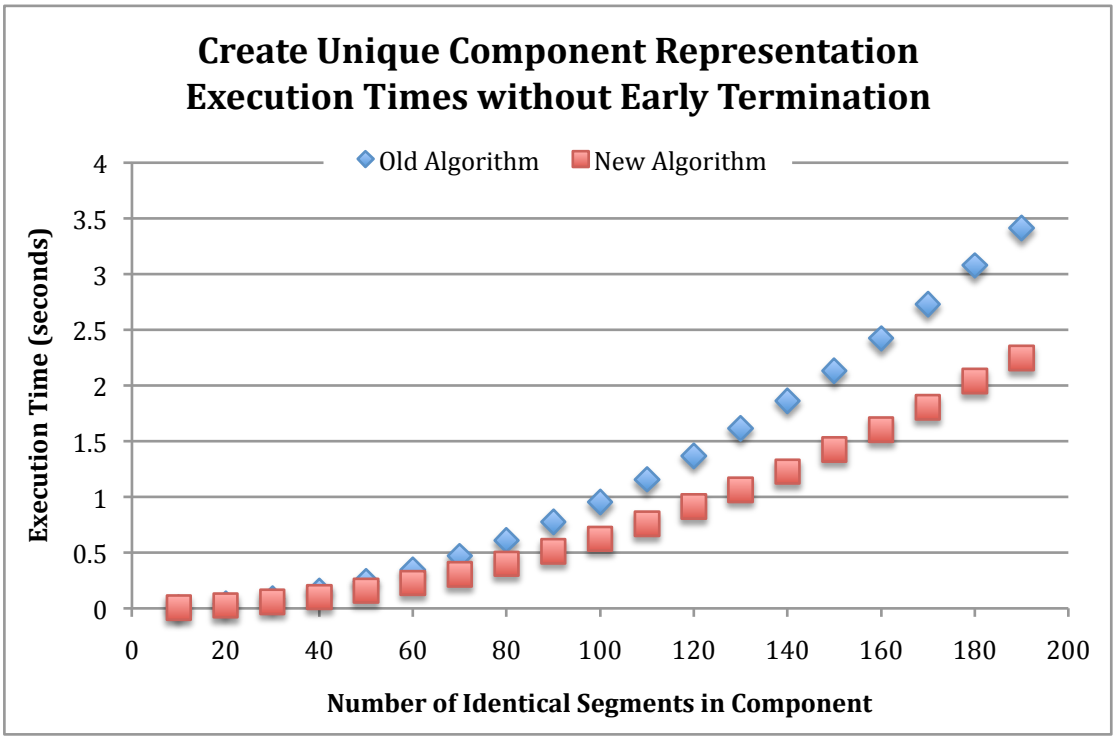


Figure 2. Creating unique component representations for components with identical segments, without using early termination.

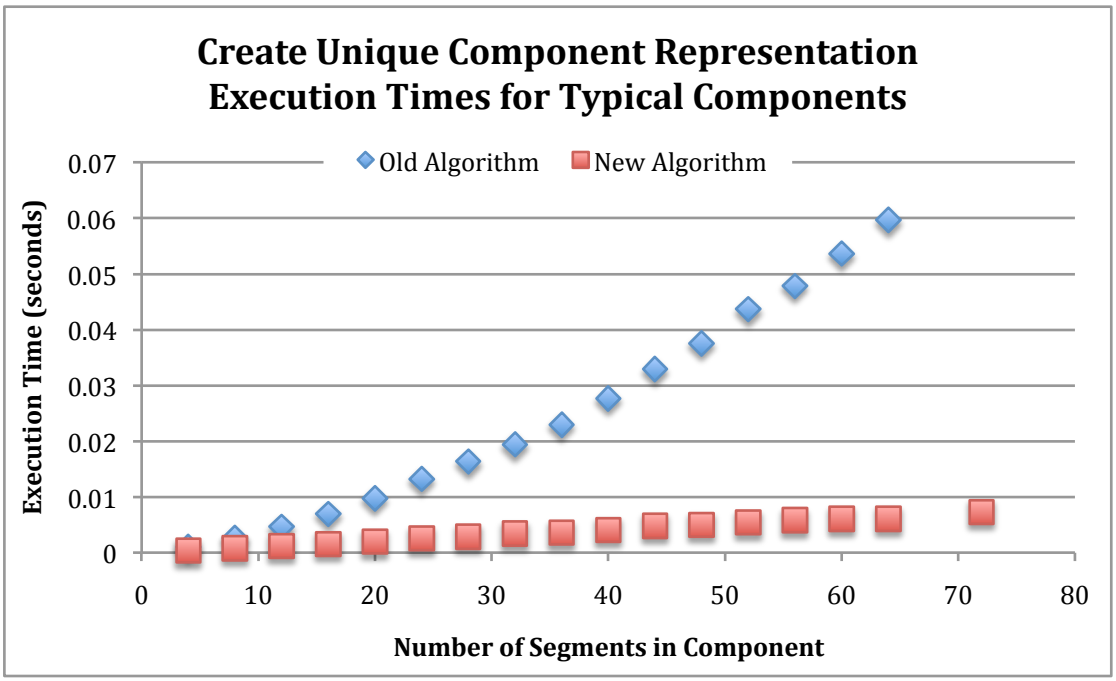


Figure 3. Creating unique component representations for components created during cube building. Early termination is used.

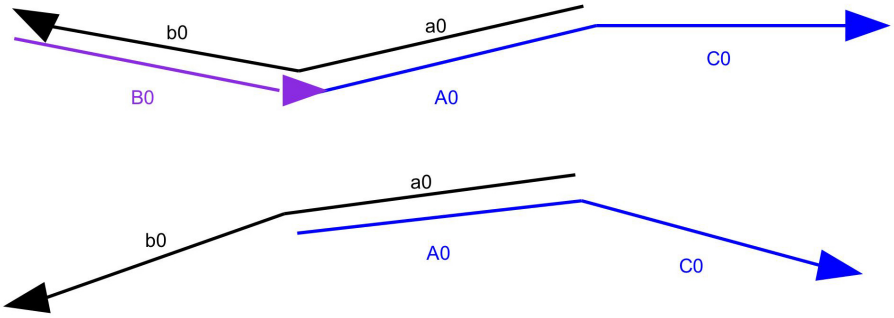


Figure 2.5: Two similar components.

2.2 Component

A component is a collection of segments that are all connected. When a separating transition such as *separate* or *displace* is performed on segments of one component, the component invariant of holding only associated segments that are connected to each other, must be maintained. In the case of a separating transition, when one component splits into two, new component instances are formed.

The problem of finding a unique representation of a component is non-trivial, and it is the process that consumes the majority of execution time in the end product. The complexity of the problem might be better understood when considering the definition of uniqueness. **Only** identical components hold identical representations, and identical components **must** hold identical representations.

2.2.1 Unique Representation

In order to achieve a unique representation of a component, the associated segments must be listed in a unique order, and information about their connectivity must be included. If the segments are not ordered uniquely then identical components might list their segments in different orders. If the connectivity of the segments is not included, then different components with segments of identical labels but with differing connectivity might return identical representations.

A unique ordering of segments can be achieved by sorting all segments, selecting the first segment in the sorted list, and using that segment's method for iterating over all its connected segments. I.e. in order to get a unique order of segments the difficulty lies in selecting one unique segment. This is also the only the hardest part of comparing two components. Once one segment from each of the two components being compared have been uniquely selected, testing for equality between the components is the same as testing

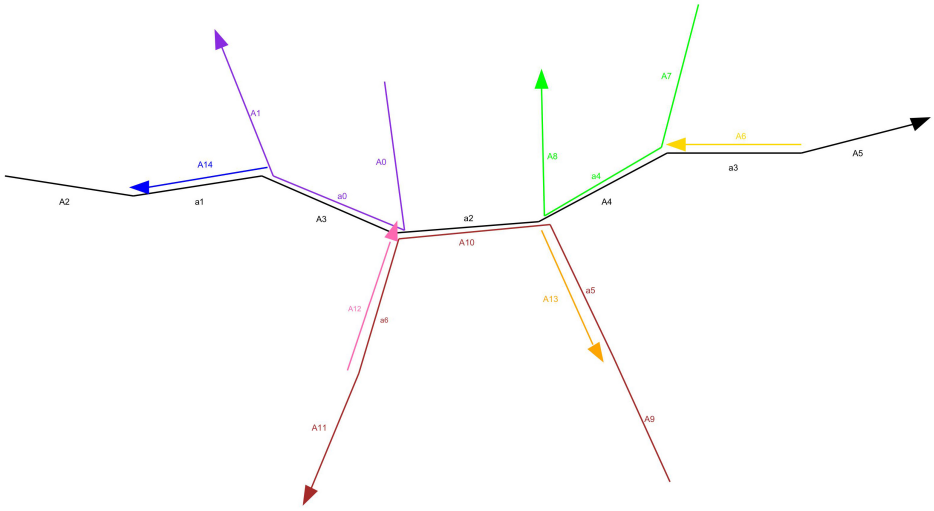


Figure 2.6: Component with alternating segment labels.

for equality between the uniquely selected segments.

The problem of sorting a list of segments is that it can be costly, as comparing two segments can be costly if they are very similar. This was previously discussed in Section 2.1.2. Especially if the component is large, and there are many similar segments in the list to be sorted, this can be a major problem. A screening process was therefore developed to reduce the size of this list.

In the segment screening process, subsets for different segment properties are formed, and segments with the appropriate properties are added to these sets. For example there is a subset for all segments with an annealed segment, and there is a subset for all segments with a particular label. An annealed segment with label `a`, and ligated at the three end, would then be added to the *annealed* subset, the *label-"a"* subset, the *ligated-at-3'-end* subset, and the *not-ligated-at-5'-end* subset. These subsets can then further be combined by creating subsets of two or more original subsets by taking their intersection. For example such a derived subset is the subset for the annealed but not-ligated-at-3'-end segments, and the subset for the unannealed segments with label `b`. The smallest subset is then selected, and sorting is performed only on this smaller set of segments.

If only one segment property is considered when examining the component shown in Figure 2.6, then there are at least as many as seven of this same property. There are for example seven segments with label `a`, there are eight unannealed segments, and eight segments with no ligated segment at the 3' end. When looking at more properties at the same time however only segment `A3`, `A4`, and `A10` have the property of having label `A`, the property

of being annealed, and being ligated at both the 3' end and the 5' end. This means that sorting only need to be performed on these 3 segments to select a unique segment for the component.

If there are multiple subsets of the smallest subset size, then the subset is chosen by a precedence rule. Also worth noting is that as the number of ways allowed to make the selection for the subset combination increase, the number of resulting subsets from intersections will increase exponentially, according to Equation (2.1). For example if only combinations by selecting 2 subsets from 10 available, is made, the number of resulting combinations is 45 plus the original 10. If all combinations from combining 3 from 10 is also made, there will be an additional 120 combinations (assuming there are no empty intersections).

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{2.1}$$

To describe the connectivity of the segments, they are all given an index so that two segments with identical labels can be distinguished from each other. A segment index is easily obtained by looking up the index of the segment in the uniquely ordered list of segments. A unique component representation is then achieved by iterating over all segments in a unique order, and for every segment describing it's connectivity to its closest neighbors, i.e. any annealed segment, and any segment ligated at 3' end and 5' end.

2.2.2 Perform Node-Initiated-Transitions

Some components can be described to consist of configurations that are not very likely to remain for very long. In order to create a realistic simulation of the behavior of DNA, a method to perform all very likely and spontaneous transitions in a component, is needed. For example, two ligated segments that are each other's inverses, are assumed to instantaneously fold up and anneal to each other.

There are three different so called *node-initiated-transitions*. These are the *displace*, the *exchange*, and the *node-initiated-anneal* (just described). What they all have in common is that the point where the transition starts is located at a node. Since the bases on the DNA strand are located right next to each other, they are assumed to instantaneously create a link. In the case of the *node-initiated-anneal*, when two unannealed segments are ligated to each other, and they are each other's inverses, it is assumed that they instantaneously anneal to each other. The *displace* and the *exchange* transition can be described similarly.

When a potential displace transition is discovered it is assumed take place instantaneously. There are two things that can happen next. Either the displaced segment did not have a secondary connection to the component,