Project Report: Finding maximal common joins in a DAG

Jim Newton

November 17, 2016

Abstract

Given a directed acyclic graph (DAG) and two arbitrary nodes, find maximal common joins of the two nodes. In this technical report I suggest an algorithm for efficiently calculating the minimal set of nodes which derive from a pair of nodes.

1 Motivation

In Common Lisp type hierarchies including CLOS class hierarchies form a directed lattice with the least specific class, `T`, being the top of the lattice and the most specific class, `NIL`, being the bottom. The lattice is can be illustrated as in Figure 1 with has arrows pointing downward from each class to its direct superclasses. If otherwise a class has no arrow leaving it, then there is an arrow to it from `NIL`. Arrows point upward from most specific to least specific classes.

Given two classes, we wish to find the least specific common subclasses.

2 Example

Looking at the DAG shown in Figure 1 we can see that classes `S7`, `S11`, `S14`, `S15` as well as a few others are subclasses of both `X` and `Y`. However, what is the set S such that all elements of S are common subclasses of `X` and `Y` but S contains no distinct elements A and B such that $A \subset B$? That would be the set containing exactly `S7`, `S11`, and `S16`.

3 Algorithm

The following is a description of an efficient algorithm for finding such a minimal set. The algorithm proceeds in two phases: down-traversal and up-traversal.

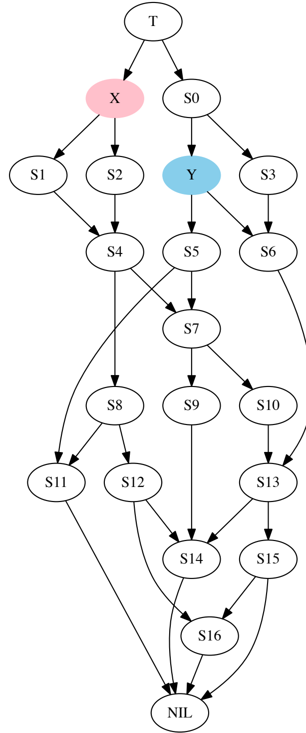


Figure 1: Example CLOS class graph

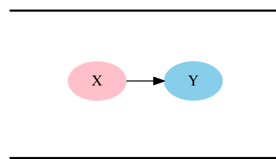
3.1 Phase One: down traversal

The DAG will be traversed in a particular order. The traversal algorithm resembles a breadth-first-search algorithm, but is initialized differently which leads to the nodes of the graph being visited in an order different than breadth first.

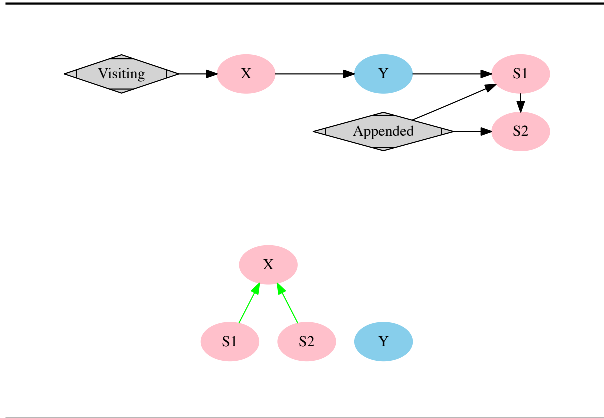
The down-traversal phase proceeds as follows:

- Initialize a queue with the starting nodes `X` and `Y`.
- Assign the color pink for `X` and its children and blue for `Y` and its children.
- Visit each node in the queue starting at the beginning. The queue will grow while traversing.
  - Visit each non-NIL child.
    1. Add a back-pointer from child to parent.
    2. If the child is not already somewhere in the queue, add it to the end.

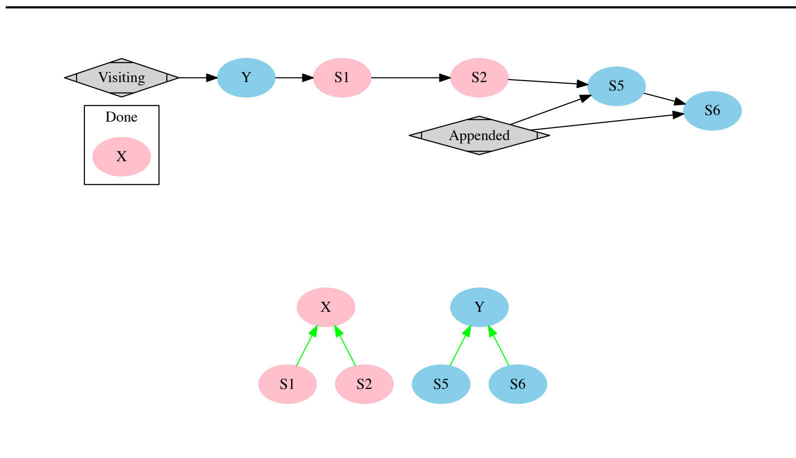
3. If the child is not yet colored, color it the same as its parent.
4. If the child is already colored but different than its parent, change its color to lavender, and remember it in the lavender list.



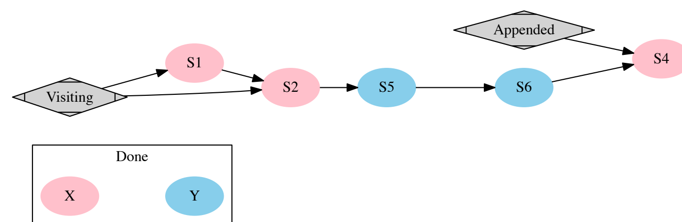
Down traversal initial state: We'll use the DAG shown in Figure 1 as an example. Initialize a queue with the starting nodes `X` and `Y`.

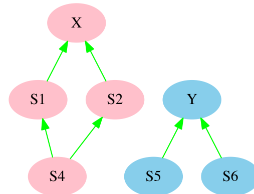


Now proceed visiting the nodes in the queue in order, starting with `X`. Append each child of the node to the end of the list. The direct children of `X` are `S1` and `S2`, so `S1` and `S2` are colored pink, the same as `X`. Construct respective back-pointers from `S1` and `S2` to `X`.

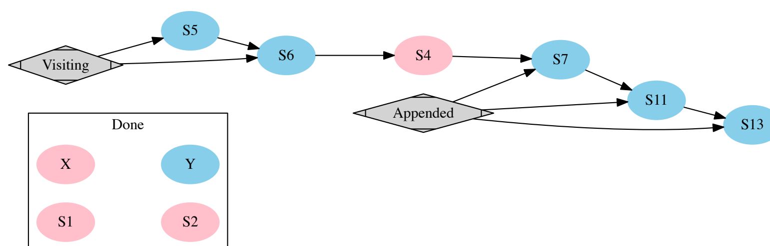


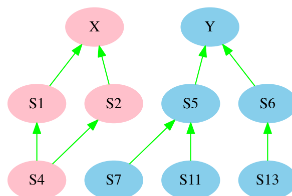
Now visit `Y`. Append each child of the node to the end of the list: `S5` and `S6` each with a back-pointer to `Y` and each inheriting the set of root nodes `Y` from its parent. At this point the DAG has been partially breadth-first traversed as shown.



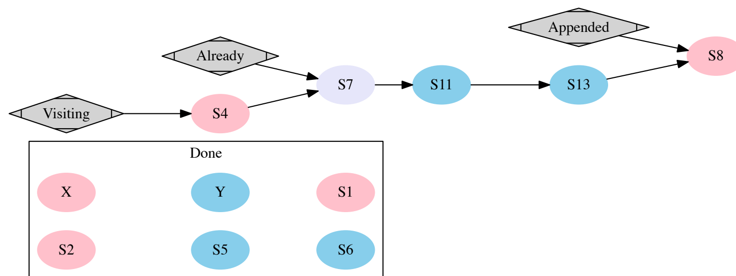


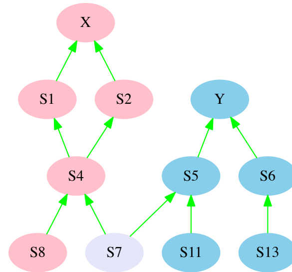
Now visit `S1`. Append the single child `S4` to the end of the queue, with a back-pointer to `S1`. `S4` inherits the color, pink, of its parent, signifying that it is a descendant of `X`.
Now visit `S2`. Its single child, `S4` is already in the list, so we simply create a back-pointer from `S4` to `S2`. The color of `S4` is already pink, and the color of its parent `S2` is also pink, so there is no need to change the color of `S4`.



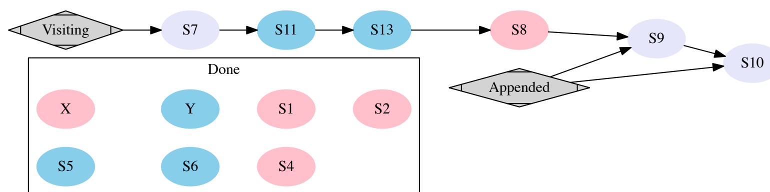


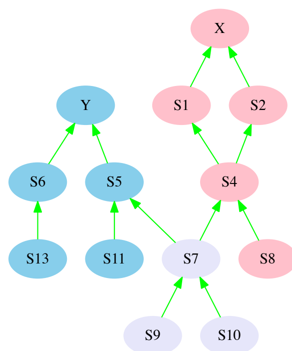
Next visit `S5` and `S6` in turn. The children of `S5` are `S7` and `S11`. The child of `S6` is `S13`. The nodes `S7`, `S11`, and `S13` are added with appropriate back-pointers and root sets from their respective parents. They are colored blue, the same as their respective parents, indicating that they are descendants of `Y`.



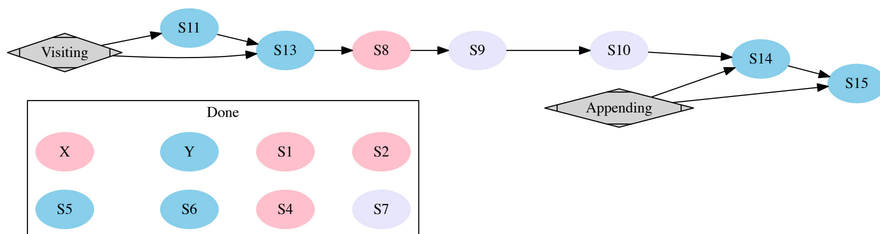


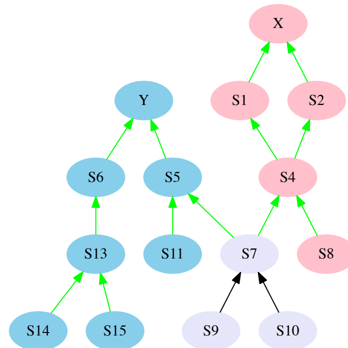
Next visit S4. The children of S4 are S7 and S8. S7 is already in the queue, but we add S8 to the end of the queue. Add a back-pointer from S7 to S4 and one from S8 to S4. Since S7 now has back-pointers both to pink and blue nodes (S4 and S5) we change its color to lavender. Since S8 is newly added, then it gets the color pink of the node being visited.



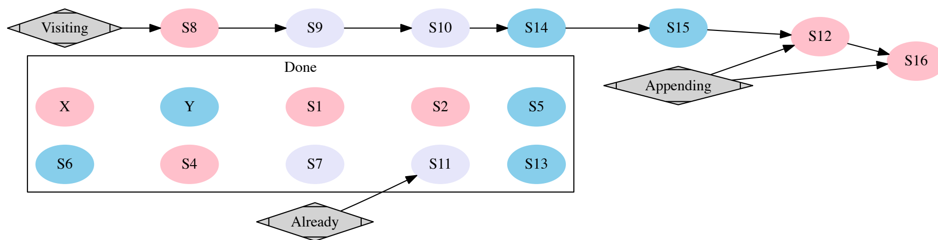


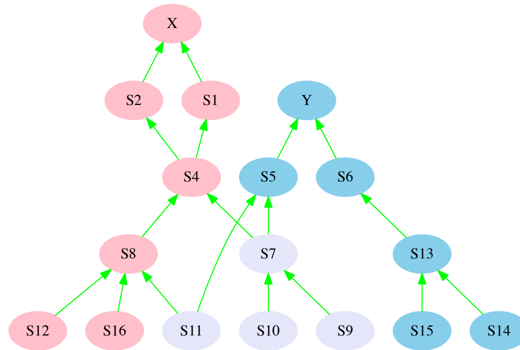
Next in the queue is `S7`. `S7` has two children `S9` and `S10`. Add them to the end of the queue, and color them them lavender, the same as `S7`. Add back-pointers from `S9` and `S10` respectively to `S7`.



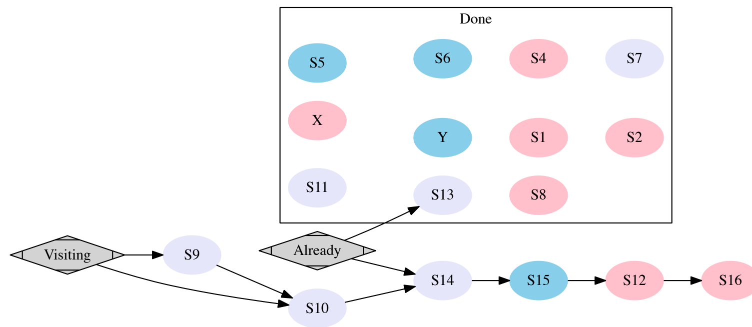


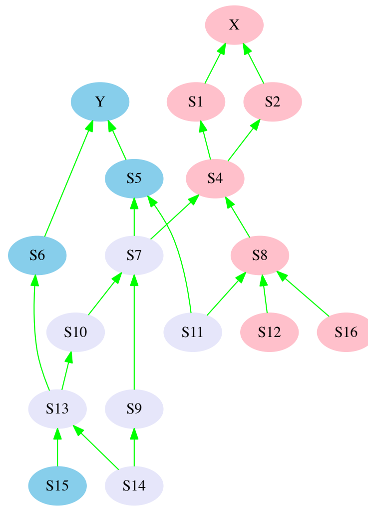
Next visit `S11` and `S13`. Their only child of `S11` is `NIL`, so there is nothing to do for it. The children of `S13` are `S14` and `S15`. We add `S14` and `S15` to the queue, with appropriate back-pointers. `S14` and `S15` are colored blue, the same as `S13`.



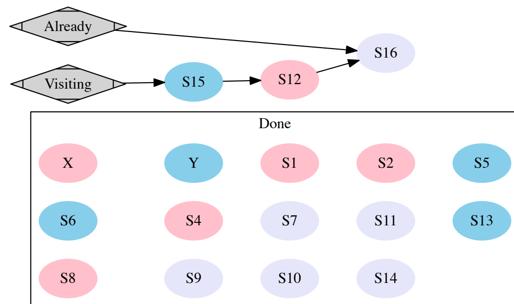


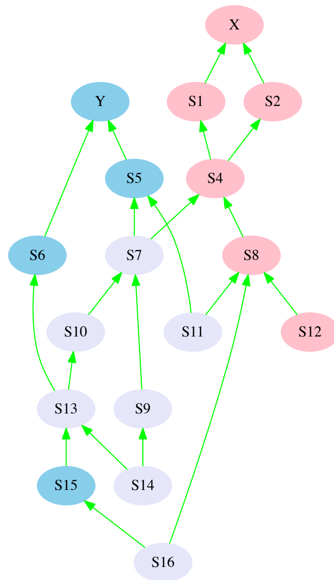
Now visit `S8`. `S8` has three children: `S11`, `S12` and `S16`. `S11` is already in the queue with color blue. Since the color of `S8` is pink, we change the color of `S11` to lavender. We add the other two nodes `S12` and `S16` to the queue.



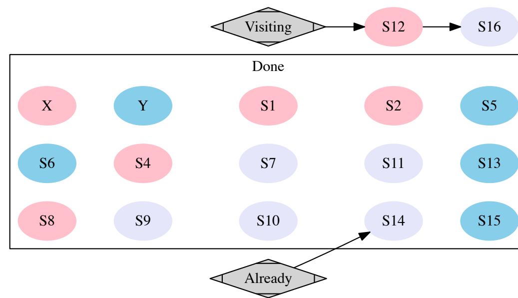


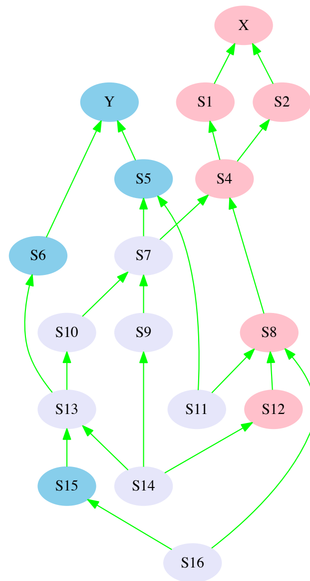
Next we visit `S9` and `S10` which are both colored lavender. Its their children, `S13` and `S14` are already in the queue, so we change their color to lavender to their respective parents `S10` and `S19`, and create the back-pointers.
One thing to notice at this point is that the blue node `S15` has a lavender parent, but remains colored blue as we are not treating it in this step of the iteration.





Now visit `S15`. Its child is `S16` which is already in the queue and is colored pink. Since the color of `S15` is blue, we must change the color of `S16` to lavender. We also add a back-pointer from `S16` to `S15`





Next we visit `S12` which has one child `S14`. `S14` is already in the queue and is already colored lavender so we don't need to change its color. We simply add a back-pointer from `S14` to `S12`.

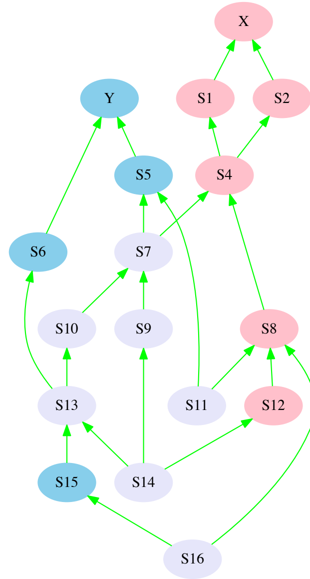


Figure 2: Final graph of back-pointers

Finally we visit `S16`. It has no children so there is nothing to do. This completes the graph defined by the back-pointers.

One thing to recognize that this graph (Figure 2) could look different depending on the order that the child nodes are treated. However, in any case you will have at this point a graph with some lavender nodes, otherwise the classes `X` and `Y` would have no common non-NIL subclasses.

Also notice, that not all common subclasses of `X` and `Y` are colored lavender. For example, `S15` remains blue, simply as an artifact of the order the graph was traversed. This is not a problem.

3.2 Phase Two: up traversal

The graph shown in Figure 2 contains some lavender nodes. These nodes form a subset of the common subclasses of `X` and `Y`. To find the least specific of these these, we simply eliminate any lavender node for which another lavender node is reachable from it via the back-pointers.

For example, starting at `S16`, we find that `S13` is reachable by tracing the back-pointers. So we eliminate `S16` from the candidates. To determine this efficiently, we can perform a breadth-first-search starting at node `S16` in 2. We terminate the search as soon as we find another lavender node. If we find no lavender node, the search will continue until the graph is exhausted. Here is an

example BFS (breadth-first-search), the results may differ according to which order the back-pointers from a given node are traversed.

1. S16 S8 S15
2. S8 S15 S4
3. S15 S4 S13

The BFS terminates after step 3 because `S13` is encountered whose color is lavender.

Likewise, `S13` can be eliminated because `S10` is reachable from it.

However, `S7` and `S11` have no lavender node reachable via their back-pointers.

1. S11 S4 S5
2. S4 S5 S1 S2
3. S5 S1 S2 Y
4. S1 S2 Y X
5. S2 Y X
6. Y X
7. X

From this we conclude the least specific common subclasses of `X` and `Y` are exactly `S7` and `S11`.

4 Example implementation

First define a useful macro, `unionf`, which allows us to use a syntax like `(unionf place value)` which is equivalent to `(setf place (union place value))`.

```
(define-modify-macro unionf (&rest args) union)
```

Define a generic function, so we can handle class specializers isolated from equivalence specializers. Methods of this generic function will compute and return a list of specializers (classes for example) which inherit from both spec1 and spec2. E.g., if spec1=class-A and spec2=class-B, then calculate a list of classes inheriting from both A and B. But if C and D both inherit from A and B, but C also inherits from D, then omit C in the return list.

```
(defgeneric specializer-intersections (spec1 spec2))
```

Define a function to calculate the read-accesser from a plist.

```
(defun getter (key)
  #'(lambda (obj)
      (getf obj key)))
```

The method for two classes does a breadth first search down the class hierarchy towards `NIL`. Each element of the BFS queue is a plist with three fields.

:class class in the sub-tree of either class1 or class2 (or both)

:roots subset of class1,class2 indicating which root classes are eventually superclasses of the :class field of this plist.

:supers list of plists, back-pointers. If classX has sub as direct-subclass then the node with :class = sub has the node with :class = classX in its :supers list.

After the BFS finishes, `(car queue)` remains a list of plists corresponding to the union of the sub-trees of class1 and class2. If any class appears on both sub-trees, then only one corresponding node is found in `(car queue)` but it has field :roots containing class1 and class2. These correspond to the lavender nodes in Figure 2.

Finally filter the list of plists. Find all the plists which are descendants of both class1 and class2, but which have no parent node which are also; both descendants thereof. Return this list of classes.

```
(defmethod specializer-intersections ((class1 class)
                                      (class2 class))
  (let ((both-roots (list class1 class2))
        (queue (tconc nil
                      (list :class class1
                            :roots (list class1)
                            :supers nil)
                      (list :class class2
                            :roots (list class2)
                            :supers nil))))
    (dolist (node (car queue))
      (dolist (sub (class-direct-subclasses (getf node :class)))
        (let ((found (find sub (car queue) :key (getter :class))))
          (cond
            (found
             (pushnew node (getf found :supers))
             (unionf (getf found :roots) (getf node :roots)))
            (t
             (tconc queue (list :class sub
                                :roots (getf node :roots)
                                :supers (list node))))))))

    (mapcar (getter :class)
```

```
          (setof node (car queue)
            (and (subsetp both-roots (getf node :roots))
                 (not (exists super-node (getf node :supers)
                        (subsetp both-roots
                                 (getf super-node :roots))))))))
```

Special handling of equivalence specializers.

```
(defmethod specializer-intersections ((eql1 eql-specializer)
                                      (class2 class))
  (when (typep (eql-specializer-object eql1) class2)
    (list eql1)))

(defmethod specializer-intersections ((class1 class)
                                      (eql2 eql-specializer))
  (when (typep (eql-specializer-object eql2) class1)
    (list eql2)))

(defmethod specializer-intersections ((eql1 eql-specializer)
                                      (eql2 eql-specializer))
  (when (or (eql eql1
                 eql2)
            (eql (eql-specializer-object eql1)
                 (eql-specializer-object eql2)))
    (list eql1)))
```

5 Conclusion

In this report I have presented and illustrated an algorithm to find the set of least specific common children of two given nodes in a directed acyclic graph. The algorithm is useful particularly in CLOS when finding the set of least specific subclasses of two given classes. In addition to the illustration, I have presented a sample implementation written in Common Lisp.

6 Acknowledgments

Special thanks to Alexandre Duret-Lutz (EPITA/LRDE) with whom I brainstormed several times working out the approach and some of the detail of this algorithm.