Topological Ordering

```
class Graph{
  Set<Node> nodes = new ...;

  List<Node> topologicalOrder(){
      ...
  }
  Node sourceNode(){
      ...
  }
  int predCount(){
      ...
  }
  ...
}
class Node{ ...}
```

```
public List<Node> topologicalOrder(){
   List<Node> orderedNodes = new ...;
   while(true){
      Node source = sourceNode();
      if(source == null) break;
      orderedNodes.add(source);
==>   nodes.remove(source);    <==
   }
   if(nodes.size() > 0 ){
      throw new RuntimeException();
   }
   return new orderedNodes;
}
```

```
public Node sourceNode(){
   for (Node node : nodes) {
      if (predCount(node) == 0){
         return node;
      }
   }
   return null;
}
```

```
public int predCount(Node node){
  int predCount = 0;
   for (Node n : nodes) {
      if(n.getSuccessors().contains(node)){
         predCount++;
      }
   }
   return predCount;
}
```

Optimization Opportunity (1)

The method **`sourceNode()`** is invoked several times, in the control flow of an invocation of **`topologicalOrder()`**, with the same implicit *this* argument destructively updated by the statement **`nodes.remove(n)`** between consecutive invocations of **`sourceNode()`**.

Dynamizing **sourceNode()** (1)

It is sound to memoize/cache **sourceNode()** as long as we provide a maintenance advice that restores cache consistency after the execution of **nodes.remove(n)**.

One approach to restore cache consistency is to use an incrementalized/dynamized version of **sourceNode()** under **nodes.remove(n)**.

Dynamizing **sourceNode()** (2)

```
Node sourceNode_cache = null; <== cache
public Node sourceNode(){ <== memoized
  if(sourceNode_cache == null){
     ... <= computes source nodes
  }
  return sourceNode_cache;
}
public void maintainSourceNodeUnderNodeRemoval(Node
removed){
     ... <== maintenance
}
```

Dynamizing **sourceNode()** (3)

```
Node sourceNode_cache = null; <== cache
public Node sourceNode(){ <== memoized
  if(sourceNode_cache == null){
    for (Node node : nodes) {
      if (predCount(node) == 0){
          sourceNode_cache = node;
           break;
      }
    }
  }
  return sourceNode_cache;
}
public void maintainSourceNodeUnderNodeRemoval(Node removed){
   ... <== maintenance
}
```

How should we change what `sourceNode()` returns if a node was removed from `nodes`?

Dynamizing **sourceNode()** (4)

```
  Node sourceNode_cache = null; <== cache
  public Node sourceNode(){ <== memoized
    if(sourceNode_cache == null){
      for (Node node : nodes) {
        if (predCount(node) == 0){
            sourceNode_cache = node;
            break;
        }
      }
    }
    return sourceNode_cache;
}
public void maintainSourceNodeUnderNodeRemoval(Node removed){
   ... <== maintenance
}
```

We need to **generalize** `sourceNode()` before we can dynamize it. Why?

Generalizing **sourceNode()**

```
Set<Node> sourceNode_cache = null; <==
public Node sourceNode(){
  if(sourceNode_cache == null){
    sourceNode_cache = new ... ; <==
    for (Node node : nodes) {
      if (predCount(node) == 0){
          sourceNode_cache.add(node); <==
      }
    }
  }
  return sourceNode_cache.get(0);
}
public void maintainSourceNodeUnderNodeRemoval(Node removed){
   ... <== maintenance
}
```

We **generalize** `sourceNode()` to compute all source nodes. Essentially, we let it finish!

We used a `Set` because repetition is not allowed and ordering doesn't matter. Why?

Dynamizing **sourceNode()** (5)

```
Set<Node> sourceNode_cache = null;
public Node sourceNode(){
  if(sourceNode_cache == null){
    sourceNode_cache = new ... ;
    for (Node node : nodes) {
      if (predCount(node) == 0){
          sourceNode_cache.add(node);
      }
    }
  }
  return sourceNode_cache.get(0);
}
public void maintainSourceNodeUnderNodeRemoval(Node removed){
   ... <== ???
}
```

How should we change what the generalized `sourceNode()` returns if a node was removed from `nodes`?

Dynamizing **sourceNode()** (6)

- If the removed nodes is in `sourceNode_cache`, it should be removed.
- If a node "n" is removed from `nodes`, predCount(node) will go down for "n"'s successors and might become 0. These nodes need to be added to `sourceNode_cache`.

Dynamizing **sourceNode()** (7)

```
  public void maintainSourceNodeUnderNodeRemoval
(Node removed){
    sourceNode_cache.remove(removed);
    for(Node succ: removed.getSuccessors()){
      if(predecessorsOf(succ) == 0){
        sources.add(succ);
      }
    }
  }
```

Dynamizing **sourceNode()** (8)

```
 public List<Node> topologicalOrder(){
   List<Node> orderedNodes = new ...;
   while(true){
       Node source = sourceNode();
       if(source == null) break;
       orderedNodes.add(source);
==>    nodes.remove(source);
==>    maintainSourceNodeUnderNodeRemoval(source);
   }
   if(nodes.size() > 0 ){
        throw new RuntimeException();
   }
   return new orderedNodes;
}
```

Optimization Opportunity (2)

The method **predCount(Node)** is invoked several times, in the control flow of an invocation of **topologicalOrder()**, with the same implicit *this* argument destructively updated by the statement **nodes.remove(n)** between consecutive invocations of **predCount(Node)**.

However, not all invocations have the same **Node** argument. Would memoization alone be enough to gain the best performance?

Reordering the Computation

- Instead of lazily computing predCount for a particular node on demand, we can eagerly compute predCount for all nodes in the graph.
- What does that save?

Reordering the Computation(2)

```
public int predCount(Node node){
   int predCount = 0;
   for (Node n : nodes) {
      if(n.getSuccessors().contains(node)){
         predCount ++;
      }
   }
   return predCount;
}
```

Can we compute the predCount of all nodes together more efficiently than computing them one by one?

Reordering the Computation(3)

- Computing the predCount of one node requires a complete graph traversal.
- Computing predCount of any number of nodes can be done by a single graph traversal as well.
- Amortizing the cost of graph traversal.

Reordering the Computation(3)

```
public int predCount(Node node){
   ...
   return x;
}
```

```
Map<Node,Integer> predCount_cache = new ...;
public int predCount(Node node){
  if(!predCount_cache.contains(node)){
     predCount_cache = eagerPredCount();
  }
  return predCount_cache.get(node);
}
public Map<Node,Integer> eagerpredCount(){
 ...
}
```

Eager PredCount()

- Wrap PredCount with a loop that goes over all nodes in the graph.
- Distribute/Push that loop inside.
- Simplify.

Eager PredCount() (2)

- Wrap PredCount with a loop.

```
 public int predCount(Node node){
    ...
    return x;
 }

 public Map<Node,Integer> eagerpredCount(){
   Map<Node,Integer> returnVal = new ...
==> for(Node node : nodes){
      ...
      returnVal.put(node, x);
   }
 return returnVal;
```

Eager PredCount() (3)

- Filling in the dots.

```
   public Map<Node,Integer> eagerpredCount(){
     Map<Node,Integer> returnVal = new ...
 ==> for(Node node : nodes){
      int predCount = 0;
       for (Node n : nodes) {
         if(n.getSuccessors().contains(node)){
          predCount ++;
        }
        returnVal.put(node, predCount);
     }
   return returnVal;
}
```

Eager PredCount() (4)

- Distribute/Push that loop inside.

```
    public Map<Node,Integer> eagerPredCount(){
       Map<Node,Integer> returnVal = new ...
       Map<Node,Integer> predCount = new ...
 ==> for(Node node : nodes){
       predCount.put(node,0);
       }
 ==> for(Node node : nodes){
         for (Node n : nodes) {
           if(n.getSuccessors().contains(node)){
             predCount.put(node, predCount.get(node)++);
           }
       }
 ==> for(Node node : nodes){
         returnVal.put(node, predCount.get(node));
       }
    return returnVal;
}
```

Eager PredCount() (5)

- Distribute/Push that loop inside.

```
    public Map<Node,Integer> eagerPredCount(){
       Map<Node,Integer> returnVal = new ...
       Map<Node,Integer> predCount = new ...
 ==> for(Node node : nodes){
       predCount.put(node,0);
       }
      for (Node n : nodes) {
==>    for(Node node : nodes){
         if(n.getSuccessors().contains(node)){
           predCount.put(node, predCount.get(node)++);
        }
       }
 ==> for(Node node : nodes){
        returnVal.put(node, predCount.get(node));
      }
    return returnVal;
}
```

O (n^2)

Eager PredCount() (6)

- Simplify.

```
    public Map<Node,Integer> eagerPredCount(){
       Map<Node,Integer> returnVal = new ...
       Map<Node,Integer> predCount = new ...
 ==> for(Node node : nodes){
       predCount.put(node,0);
       }
      for (Node n : nodes) {
==> for(Node node : n.getSuccessors()){
          predCount.put(node, predCount.get(node)++);
        }
       }
 ==> for(Node node : nodes){
        returnVal.put(node, predCount.get(node));
      }
    return returnVal;
}
```

O(n+m)

Eager PredCount() (7)

- Simplify.

```
    public Map<Node,Integer> eagerPredCOunt(){
      Map<Node,Integer> predCount = new ...
 ==>  for(Node node : nodes){
       predCount.put(node,0);
       }
      for (Node n : nodes) {
 ==>   for(Node node : n.getSuccessors()){
         predCount.put(node, predCount.get(node)
++);
       }
      }
    return predCount;
}
```

O(n+m)

Eager PredCount() (3)

- Instead of Map<Node,Integer> can have a field in Node.

```
   public void eagerPredCount(){
==> for(Node node : nodes){
      node.predCount = 0;
    }
    for (Node n : nodes) {
  ==> for(Node node : n.getSuccessors()){
        node.predCount ++;
      }
    }
}
```

Dynamizing **predCount()** (1)

It is sound to memoize **predCount()** as long as we maintain the consistency after the execution of **nodes.remove(n)**. One approach to restore cache consistency is to use an incrementalized/dynamized version of **predCount()** under **nodes.remove(Node)**.

Dynamizing **predCount()** (2)

```
public int predCount(Node node){
int predCount = 0;
for (Node n : nodes) {
   if(n.getSuccessors().contains(node)){
      predCount ++;
   }
}
return predCount;
}
```

How should we change what this method returns if a node was removed from nodes?

Dynamizing **predCount()** (3)

- If a node is removed from the graph, it can no longer have predecessors in that graph.
- If a node is removed from the graph, all of it's successors are no longer going to have the removed node as a predecessor.

Dynamizing **predCount()** (4)

```
  public void maintainPredCountUnderNodeRemoval(Node
removed){
    predCount_cache.remove(removed);
    for(Node succ: removed.getSuccessors()){
          predCount_cache.put(succ, predCount_cache.get
(succ))--;
    }
  }
```

Dynamizing **sourceNode()** (5)

```
 public List<Node> topologicalOrder(){
   List<Node> orderedNodes = new ...;
   while(true){
       Node source = sourceNode();
       if(source == null) break;
       orderedNodes.add(source);
==>    nodes.remove(source);
==>    maintainPredCountUnderNodeRemoval(source);
   }
   if(nodes.size() > 0 ){
        throw new RuntimeException();
   }
   return new orderedNodes;
}
```

Combining Both Optimizations

- Both optimizations add maintenance code right after **nodes.remove(n)**.
- When combining them which one should come first? Why?

```
==>    nodes.remove(source);
==>    maintainSourceNodeUnderNodeRemoval(source);
==>    maintainPredCountUnderNodeRemoval(source);
```

Or

```
==>    nodes.remove(source);
==>    maintainPredCountUnderNodeRemoval(source);
==>    maintainSourceNodeUnderNodeRemoval(source);
```

Combining Both Optimizations

- Both optimizations add maintenance code right after **nodes.remove(n)**.
- When combining them which one should come first? Why?

```
==>    nodes.remove(source);
==>    maintainPredCountUnderNodeRemoval(source);
==>    maintainSourceNodeUnderNodeRemoval(source);
```

- Because sourceNode calls predCount.