Optimize Your Programs, Elegantly

By: Ahmed Abdelmeged
Joint work with: Karl Lieberherr
Agenda

- Introduction.
- Dijkstra's Shortest Paths.
- Topological Ordering.
You can make your programs more elegant by

- Having “components that influence other components”.
- BTW, these are called “aspects”.
- AspectJ is a general purpose programming languages for developing aspects.
- You can also simulate some what AspectJ aspects do using inheritance:
  - A subclass is an influenced version of its superclass. (Java)
  - Mixins, Traits produce influenced versions of classes. (Scala)
Figuring Out Aspects

- We recommend that you figure out aspects right from the requirements rather than refactoring your program into aspects.
- Efficiency is one such requirement that can be implemented as an aspect.
Dijkstra’s Shortest Paths

Given a weighted Graph \((g)\) and a source node \((s)\), find the shortest path from \(s\) to all nodes in \(g\).
Dijkstra's Shortest Paths

To each node, maintain it's best known distance from source and its predecessor on the shortest path from source.

Maintain a list of active nodes. Initially = \{S\}.

Maintain a list of final nodes. Initially = {}
Dijkstra’s Shortest Paths

- Select the closest to source active node (v).
- Update the best known distance as well as predecessors of v’s successors that are not marked as final.
- Mark v as final.
Dijkstra’s Shortest Paths

- Select the closest to source active node (v).
- Update the best known distance as well as predecessors of v’s successors that are not marked as final.
- Mark v as final.
Dijkstra’s Shortest Paths

- Select the closest to source active node (v).
- Update the best known distance as well as predecessors of v’s successors that are not marked as final.
- Mark v as final.
Dijkstra's Shortest Paths

- Select the closest to source active node (v).
- Update the best known distance as well as predecessors of v's successors that are not marked as final.
- Mark v as final.
To node we add:

```java
private int distance = Integer.MAX_VALUE;
private Node predecessor = null;
private boolean finalized = false;

public void init(boolean isSource) {
    finalized = false;
    predecessor = null;
    distance = isSource ? 0 : Integer.MAX_VALUE;
}

public boolean isFinal() { return finalized; }

public void markAsFinal() { finalized = true; }

public boolean isReachable() {
    return distance != Integer.MAX_VALUE;
}

public boolean isActive() {
    return !isFinal() && isReachable();
}

public Maybe<Path> getShortestPath() {
    ...
}

public void updateDistances() {
    for (Edge e : getOutgoingEdges()) {
        Node v = (Node) e.getTarget();
        if (!v.isFinal()) {
            int alt = distance + e.getLength();
            if (alt < v.distance) {
                v.distance = alt;
                v.predecessor = this;
            }
        }
    }
}

@Override
public int compareTo(Node o) {
    return distance - o.distance;
}
```
Note that:

Instead of maintaining the set of final nodes, we maintain a finalized bit for each node. Otherwise, we need to pass too much information (the list of all final nodes) to Node.updateDistances().

distance, finalized and predecessor are only meaningful during an invocation of dijkstra(..).
public Map<Node, Maybe<Path>> dijkstra(Node source) {
    // Initialize Dijkstra node extension
    for (Node n : getNodes()) {
        n.init(n == source);
    }
    // Compute shortest paths
    for (Node u = getClosestActiveNode(); u != null; u = getClosestActiveNode()) {
        u.markAsFinal();
        u.updateDistances();
    }
    // retrieve shortest paths
    Map<Node, Maybe<Path>> result = new ...
    for (Node node : getNodes()) {
        result.put(node, node.getShortestPath());
    }
    return result;
}

public Node getClosestActiveNode() {
    Collection<Node> activeNodes = getActiveNodes();
    if (activeNodes.size() == 0) {
        return null;
    } else {
        return Collections.min(activeNodes);
    }
}

private Collection<Node> getActiveNodes() {
    List<Node> active = new ...
    for (Node n : getNodes()) {
        if (n.isActive()) {
            active.add(n);
        }
    }
    return active;
}
Optimizations

Analyzing the execution traces of our implementation we discover that the two methods `getClosestActiveNode()` and `getActiveNodes()` are invoked several times with the same graph object.

We maintain a priority queue of nodes ordered by the node’s shortest known distance from source and use it to speed up `getClosestActiveNode()`.
public aspect ActiveNodes
percflow(execution(public Map<Node, Maybe<Path>> dijkstra(Node))){
    //active nodes in the graph ordered by their distance from source
    private PriorityQueue<Node> activeNodes = new ...

    after(Node n): call(public void markAsFinal()) && target(n){
        activeNodes.remove(n);
    }

    before(Node n): set(private int Node.distance) && target(n){
        if(n.isReachable()){
            activeNodes.remove(n);
        }
    }

    after(Node n): set(private int Node.distance) && target(n){
        if(n.isReachable()){
            activeNodes.add(n);
        }
    }

    Node around(): call(public Node getClosestActiveNode()){
        return activeNodes.peek();
    }
}
Why keeping the optimization separate?

- No need to change method signatures (what's the problem with changing method signatures?).
- Fewer things to keep in your head while:
  - Developing the algorithm.
  - Understanding/Maintaining the algorithm.
- Reduced scattering an tangling of optimization related code with the base program.
- Safety: Algorithm + Optimization Aspect ≈ Algorithm
Changing Method Signatures

```java
public Map<Node, Maybe<Path>> dijkstra(Node source) {
    private PriorityQueue<Node> activeNodes = new ...;
    // Initialize Dijkstra node extension
    for (Node n : getNodes()) {
        n.init(n == source);
    }
    // Compute shortest paths
    for (Node u = activeNodes.peek(); u != null;
        u = activeNodes.peek()) {
        u.markAsFinal();
        if(v.isReachable()){
            activeNodes.remove(u);
        } *=
        u.updateDistances(activeNodes);
    }
    // retrieve shortest paths
    Map<Node, Maybe<Path>> result = new ...
    for (Node node : getNodes()) {
        result.put(node, node.getShortestPath());
    }
    return result;
}
```

```java
public void updateDistances(PriorityQueue<Node> activeNodes) {
    for (Edge e : getOutgoingEdges()) {
        Node v = (Node) e.getTarget();
        if(!v.isFinal()){
            int alt = distance + e.getLength();
            if(alt < v.distance){
                if(v.isReachable()){
                    activeNodes.remove(v);
                    v.distance = alt;
                    if(v.isReachable()){
                        activeNodes.add(v);
                    } v.predecessor = this;
                }
            }
        }
    }
}
```

```java
public void init(boolean isSource, PriorityQueue<Node> activeNodes) {
    finalized = false;
    predecessor = null;
    if(isReachable()){
        activeNodes.remove(this);
    }
    distance = isSource? 0 :Integer.MAX_VALUE;
    if(isReachable()){
        activeNodes.add(this);
    }
}
```
Safety

Algorithm + Optimization Aspect ≈ Algorithm

What if you forgot one of the lines related to optimization?

Unit tests can catch the problem, but you’ll get a single bit of information.

We have a tool that monitors the execution of optimization aspects and notifies you right after the optimization aspects first derails the algorithm rather than when the algorithm finishes.
Topological Ordering

- Given a graph (g), find an ordering of its nodes such that node (a) comes before node (b) if there is a path from a to b.

- $<s, x, z, w>$?
- $<s, x, w, z>$?
- $<s, w, x, z>?$
- $<s, x, z, w>$?
public Maybe<List<Node>> topord() {
    List<Node> orderedNodes = new ...;
    for (Maybe<Node> mbNode = getSourceNode();
         mbNode.isSome();
         mbNode = getSourceNode()) {
        orderedNodes.add(mbNode.get());
        remove(mbNode.get());
    }
    if (getNumNodes() > 0) {
        return new None<List<Node>>();
    }
    return new Some<List<Node>>(orderedNodes);
}

public Maybe<Node> getSourceNode() {
    for (Node n : getNodes()) {
        if (n.getPredCount() == 0) {
            return new Some<Node>(n);
        }
    }
    return new None<Node>();
}

public int getPredCount() {
    int predCount = 0;
    for (Node n : getEnclosingGraph().getNodes()) {
        if (n.hasSuccessor(this)) predCount++;
    }
    return predCount;
}
public Maybe<List<Node>> topord() { 
    List<Node> orderedNodes = 
        new ArrayList<Node>(); 
    for(Maybe<Node> mbNode = getSourceNode(); 
        mbNode.isSome(); 
        mbNode = getSourceNode()){
        orderedNodes.add(mbNode.get()); 
        remove(mbNode.get()); 
    } 
    if(getNumNodes() > 0 ){ 
        return new None<List<Node>>(); 
    } 
    return new Some<List<Node>>(orderedNodes); 
}

public Maybe<Node> getSourceNode(){ 
    for (Node n : getNodes()) {
        if(n.getPredCount() == 0){
            return new Some<Node>(n);
        }
    }
    return new None<Node>(); 
}

public int getPredCount(){
    int predCount = 0; 
    for (Node n : getEnclosingGraph().getNodes()) {
        if(n.hasSuccessor(this)) predCount++; 
    }
    return predCount; 
}
Topological Ordering

```java
public Maybe<List<Node>> topord(){
    List<Node> orderedNodes =
        new ArrayList<Node>();
    for(Maybe<Node> mbNode = getSourceNode();
        mbNode.isSome();
        mbNode = getSourceNode()){
        orderedNodes.add(mbNode.get());
        remove(mbNode.get());
    }
    if(getNumNodes() > 0 ){
        return new None<List<Node>>();
    }
    return new Some<List<Node>>(orderedNodes);
}

public Maybe<Node> getSourceNode(){
    for (Node n : getNodes()) {
        if(n.getPredCount() == 0){
            return new Some<Node>(n);
        }
    }
    return new None<Node>();
}

public int getPredCount(){
    int predCount = 0;
    for (Node n : getEnclosingGraph().getNodes()) {
        if(n.hasSuccessor(this)) predCount++;
    }
    return predCount;
}
```
public Maybe<List<Node>> topord() {
    List<Node> orderedNodes =
        new ArrayList<Node>();
    for (Maybe<Node> mbNode = getSourceNode();
         mbNode.isSome();
         mbNode = getSourceNode()) {
        orderedNodes.add(mbNode.get());
        remove(mbNode.get());
    }
    if (getNumNodes() > 0) {
        return new None<List<Node>>();
    }
    return new Some<List<Node>>(orderedNodes);
}

public Maybe<Node> getSourceNode() {
    for (Node n : getNodes()) {
        if (n.getPredCount() == 0) {
            return new Some<Node>(n);
        }
    }
    return new None<Node>();
}

public int getPredCount() {
    int predCount = 0;
    for (Node n : getEnclosingGraph().getNodes()) {
        if (n.hasSuccessor(this)) predCount++;
    }
    return predCount;
}
Topological Ordering

```java
public Maybe<List<Node>> topord(){
    List<Node> orderedNodes =
        new ArrayList<Node>();
    for(Maybe<Node> mbNode = getSourceNode();
        mbNode.isSome();
        mbNode = getSourceNode()){
        orderedNodes.add(mbNode.get());
        remove(mbNode.get());
    }
    if(getNumNodes() > 0 ){
        return new None<List<Node>>();
    }
    return new Some<List<Node>>(orderedNodes);
}

public Maybe<Node> getSourceNode(){
    for (Node n : getNodes()) {
        if(n.getPredCount() == 0){
            return new Some<Node>(n);
        }
    }
    return new None<Node>();
}

public int getPredCount(){
    int predCount = 0;
    for (Node n : getEnclosingGraph().getNodes()) {
        if(n.hasSuccessor(this)) predCount++;
    }
    return predCount;
}
```
The graph is gone!

Mark nodes as removed rather than actually removing them.

Graph accessors need to be modified to take those marks into account during (in the control flow of) the execution of topord()

Sounds like an aspect. Described as a modification to an existing component (the graph).
getSourceNode() and getPredCount() are invoked several times with the same arguments.

Maintain the list of current source nodes. Upon removing a node

- remove it from the list.
- potentially add its successors to the list.

Maintain the count of predecessors. Upon removing a node, decrement the predecessors count of its successors.
public aspect SrcNode {
  private List<Node> Graph.sourceNodes = null;
  // Memoization + Eager initialization
  Maybe<Node> around(Graph g): execution(public Maybe<Node> getSourceNode()) && target(g) {
    if (g.sourceNodes == null) {
      g.sourceNodes = new ArrayList<Node>();
      for (Node n : g.getNodes()) {
        if (n.getPredCount() == 0) {
          g.sourceNodes.add(n);
        }
      }
    }
    if (g.sourceNodes.size() == 0) {
      return new None<Node>();
    } else {
      return new Some<Node>(g.sourceNodes.get(0));
    }
  }
  // Maintenance
  void around(Graph g, Node n): execution(public void remove(Node)) && target(g) && args(n) {
    List<Node> successorsOfToBeRemovedNode = new ArrayList<Node>(n.getSuccessors());
    proceed(g, n);
    for (Node succ : successorsOfToBeRemovedNode) {
      if (succ.getPredCount() == 0) {
        g.sourceNodes.add(succ);
      }
    }
    g.sourceNodes.remove(n);
  }
}
public aspect Peds {

    private int Node.predCount = -1;

    // Memoization + Eager initialization
    int around(Node node): execution(public int getPredCount(..)) && target(node){
        if(node.predCount == -1){
            for (Node n : node.getEnclosingGraph().getNodes()) {
                n.predCount = 0;
            }
            for (Node n : node.getEnclosingGraph().getNodes()) {
                for (Node succ : n.getSuccessors()) {
                    succ.predCount++;
                }
            }
        }
        return node.predCount;
    }

    // Maintenance
    before(Node n): execution(private void removeAllSuccessors(..)) && target(n){
        for (Node succ : n.getSuccessors()) {
            succ.predCount--;  
            }
    }
}
To Take Home

- Factor your program into components and aspects.
- Don't refactor your program into aspects, it's harder that way.
- Figure out aspects from the requirements.
- Optimizations should be factored out into aspects.