Extra Credit: Eliminating Inefficiencies with Incremental Computation

Ahmed Abdelmeged and Karl Lieberherr
NU CCIS

1 Introduction

Here are 3 algorithms where incremental computation is used to speed them up: Dijkstra’s algorithm, Kruskal’s algorithm with Union-Find, Closest Pair algorithm. Scan your text book and any algorithms you know from other classes or from COOP for other instances where inefficiency is eliminated using incremental computation. Report the algorithms you find and how the arguments change between calls and how the computation is done incrementally.

The goal of replacing inefficiency with incremental computation is to make the algorithms run faster.

When you have decided on reporting an algorithm, report it in:

/home/lieber/.www/courses/algorithms/cs4800/sp11/extra-credit/algorithms-discussed

You can only report an algorithm that is not already used. The above 3 algorithms are considered to be used.

What to turn in for each algorithm: (1) why the algorithm is inefficient and duplicates work. (2) how you change the algorithm to eliminate the inefficiency. Which computation will be done incrementally and how? (3) What is the running-time before and after.

In the following we call the incremental computation a caching aspect.

2 Development of Caching Aspects

The development of a caching aspect proceeds according to the following process

1. Identify an inefficiency in a given program:
   (a) Identify a method \( m \) that could be invoked more than once in the same run with related arguments.
   (b) Identify the relationship between the arguments \( a_1, a_2 \) of two invocations \( m(a_1), m(a_2) \) of \( m \). The relationship could be either:
      - \( a_1 \) is the same as \( a_2 \), or
      - \( a_2 \) is an updated version of \( a_1 \) updated via one or more methods from the set \( F \).
2. For each method \( f \) in \( F \), develop an incrementalized version of \( m \) under \( f \), denoted as \( m_f \). \( m_f \) must satisfy the condition \( m \cdot f = m_f \cdot m \).
Given \( m, F \) and \( m_F \), an optimization aspect can be systematically generated. The generated aspect performs two main tasks:

1. Memoize \( m \).
2. Trap the calls made to any \( f \) in \( F \) that update the arguments \( a_1 \) of one of memoized calls to \( m \) to \( a_2 \). Use \( m_f \) to compute \( m(a_2) \) from the memoized result of \( m(a_1) \). Add \( m(a_2) \) to the memoization table.

As we shall see below, there are cases where the arguments \( a_1, a_2 \) of two invocations have a more subtle relation. Both are updated versions of some third object \( b \). Inserting a call \( m(b) \) is necessary for exposing the inefficiency to the aforementioned technique.

Also, it is not always possible to develop an incrementalized version of some method \( m \) under \( f \). For example, if \( m \) is \textit{median} and \( f \) is \textit{filter}. The result of \textit{median} is not so useful in computing \( \text{median} \cdot \text{filter} \). In this case, it is possible to replace the calls to \( m \) with an alternate implementation \( n \) that relies on an auxiliary method \( \text{aux} \) that can be incrementalized. For example, \textit{median} can be replaced with a call to \textit{sort} followed by a retrieval. \textit{sort} is an auxiliary method that can be incrementalized under \textit{filter}. In fact \textit{sort} commutes with \textit{filter}.

### 3 Case Study I: Topological Ordering

Given a directed graph \( g \) it is desired to find an ordering of its node such that \( n \) comes before \( m \) in the computed ordering if \( g \) contains an edge from \( n \) to \( m \). Figure 3.1 shows a simple algorithm for computing the topological ordering of a given directed graph. The algorithm keeps removing a node with no predecessors from the graph appending it to the output list until there is no more nodes with no predecessors.

#### 3.1 Inefficiency

There are two sources of inefficiency, the first is the method \textit{Graph.sourceNodes()} which is called several times on the same graph object updated through the method \textit{Graph.remove(Node)}. The second source of inefficiency is the method \textit{Graph.predecessorsOf(Node)} which is called several times on the same graph object updated through the method \textit{Graph.remove(Node)}.
public class Graph {

List<Node> nodes = new ArrayList<Node>();

public Graph(List<Node> nodes) {
    this.nodes = nodes;
}

public List<Node> predecessorsOf(Node node) {
    List<Node> predecessors = new ArrayList<Node>();
    for (Node n : nodes) {
        if (n.successors.contains(node)) {
            predecessors.add(n);
        }
    }
    return predecessors;
}

public List<Node> sourceNodes() {
    List<Node> sources = new ArrayList<Node>();
    for (Node node : nodes) {
        if (predecessorsOf(node).size() == 0) {
            sources.add(node);
        }
    }
    return sources;
}

public Maybe<List<Node>> topologicalOrder() {
    List<Node> orderedNodes = new ArrayList<Node>();
    for (List<Node> sources = sourceNodes(); sources.size() > 0 ; sources = sourceNodes()) {
        Node n = sources.get(0);
        orderedNodes.add(n);
        remove(n);
    }
    if (nodes.size() > 0) return new None<List<Node>>();
    return new Some<List<Node>>(orderedNodes);
}

public void remove(Node n) {
    nodes.remove(n);
}
}

Fig. 1: Algorithm for Finding The Topological Ordering of a Directed Graph