15-354: Final

Thursday, December 14, 2023

Introduction

This writeup serves three purposes. First, we explain the structure of the program files and how to use the executable output by the makefile. Second, we explain the logic behind how the new idempotent iteration finding algorithm works, the space complexity of the precomputation, and the time complexity of finding an arbitrary iteration. Third, we discuss briefly the real world performance of our specific implementation and ideas for further anaylsis of this topic.

Program Outline

There are four files containing source code and header files for three of them.

`idempotent.cpp`

implements an idempotent function as a vector of outputs, where index i corresponds the function applied to i.

`naive.cpp`

implements the "squared-speedup" version of the algorithm which includes a table of the powers-of-two iterations of the supplied idempotent function.

`graph.cpp`

implements our new algorithm for solving iterations which includes a vector that for each element of the domain of the idempotent function contains a pointer to the location of the element in our transient graph structure.

`main.cpp`

implements a menu system that explores the different solutions and their performance on user specified idempotent functions.
Note that to load a specific idempotent function, it must be stored as a text file where line i of the file contains the output of the idempotent function applied to i. For example, to define $f(0) = 1, f(1) = 1, f(2) = 0, f(3) = 2$, we would make the file

```
1
1
0
2
```

Algorithm Explanation/Intuition

Throughout our description of the algorithm, we think of the idempotent function f as a digraph, with domain/codomain elements being vertices and an edge from vertex u to v if $f(u) = v$.

If we think about computing an iteration $f^i(x)$ by simply evaluating the function i times, this corresponds to starting at vertex x on the graph and traversing the next i edges (until we hit the cycle in which we can do the trick we did on the midterm to find what point on the cycle the iteration stops at). However, this method at worst takes $O(n)$ time for a domain of size n, as we may have to traverse every element of the domain.

A natural extension of this idea involves edge contraction. In this case, contracting the edge $u \to v$ results in a vertex where all edges leading into it are elements that map to u or v, and the edge leading out of it is $v \to f(v)$. This process can be repeated for $u_1 \to f(u_1) \to f(f(u_1)) \to \cdots f^k(u_1)$ to represent a large contiguous segment of the idempotent graph. We can represent the vertex left after contraction as a vector containing the elements that make up the vertex such that the ith element of the vector is $f^i(u)$ where u is the first element in the vector. We can then link these contracted vertices to each other using pointers along with the location on the vector the vertex leads to. Computing an iteration is similar to the traversing vertices in the original graph, except now we have to check if the iteration lies in a vector within one of the contracted vertices.

The main issue with this strategy is that it is possible (and very likely) that two elements map to the same element. In this case, only one edge can be contracted while keeping the representation of the chain a simple vector. In order to choose which edge we contract, we compute the length of every transient in the idempotent graph, and contract the edges among each transient in order from largest to smallest. Mechanically, if we are in the process of contracting vertices along a transient, and we intersect an already made contracted vertex, we simply have the new transient point to the old one. Since the old one is longer than or equal in length to the new one, this will intuitively decrease the number of traversals we have to make through the graph.

Implementation and Time and Space Complexity
Let the number of elements in the domain/codomain be n.
We perform the precomputation as follows:

1. Construct the reverse graph for the idempotent graph
2. Find each cycle in the idempotent graph via Floyd's algorithm and construct a node in our new graph containing all the points in the cycle
3. Use the reverse graph to find every leaf in the idempotent graph
4. For each leaf, compute the length of the transient and insert the length and leaf into a multimap
5. Starting with the largest element, perform the edge contraction along the transient and stop once we reach a node we've already constructed. We proceed to add an edge between the transient we are contracting and the node we encounter.

Step 1 can be done in $O(n)$ time as we simply go though each mapping in the idempotent function, and insert the reverse edge in an adjacentcy list.
Step 2 can also be done in $O(n)$ time if we mark vertices we've already encountered, as Floyd's algorithm is linear with respect to the sum of the transient and cycle length.
Step 3 can be done in $O(n)$ time as we simply scan the adjacentcy list for leaves.
Step 4 can be done in $O(n\log(n))$ time, as we need to insert a maximum of n elements into an ordered map.
Step 5 can also be done in $O(n\log(n))$ time, as we need to iterate through the ordered map and contract the edges (which is $O(n)$ time).
In total, this takes $O(n\log(n))$ time, which is the same as the squared-speed up solution. We need to store the nodes of the new graph and the reverse graph. The reverse graph uses $O(n)$ space, since there are n edges and we use an adjacentcy list. The new graph collectively contains vectors with every element of the domain, which takes up $O(n)$ space, and there are at most $O(n)$ vertices (in the trivial case where no edges are contracted). Thus, collectively, the precomputation uses $O(n)$ storage, which is an improvement over the squared-speed up by a log factor.

Getting an upper bound on the running time of an evaluation is slightly more difficult, as we need to construct the worse case for our algorithm. We want to find an element that requires the most traversals in our contracted graph. For an evaluation $u \to f^i(u)$ in the contracted graph to go through k traversals, we need at least k vertices in the contracted graph that instersect the path from u to $f^i(u)$. Let the k vertices be $v_1 \leftarrow v_2 \leftarrow \cdots \leftarrow v_k$, and let these vertices contain the fewest number of elements possible. We know that each of these vertices contains a leaf from the original idempotent graph, as we create these vertices by contracting edges starting at a leaf.

We claim that for any vertex, the length of the path from its leaf to the cycle the leaf leads to is at least k. Assume for sake of contradiction that this path in v_i has length less than k. Since we create these vertices in order of greatest leaf-to-cycle length, for v_i to be the ith vertex created (which we require for the order above) every vertex v_j where $j > i$ must have a leaf-to-cycle length less than or equal to the length of v_i. However, we know that there is a path of length k from the cycle to a value on v_k. Therefore, our algorithm would contract these vertices first, which is a contradiction.

We now claim that the method of constructing these v_i involves starting with v_1 being a vertex containing k elements and each other v_j where $j > 0$ points to the last element on v_{j-1} and contains $k - j + 1$ elements. This arrangement does indeed produce a valid set of vertices, as the leaf-to-cycle path of vertex v_i has a length of $k - i + 1 + i - 1 = k$, since it includes the $k - i + 1$ elements contained in v_i and the $i - 1$ elements the path intersects which are contained in the previous $i - 1$ vertices. Now consider making v_j point to an element which preceeds the last element in v_{j-1}. This would result in the leaf-to-cycle path length of v_j increasing by an amount $m > 0$. To accomodate this, we would need to add m elements to every vertex v_l where $l < j$. Since we assume that these vertices contain the fewest elements possible, this is a contradiction. In addition, by similar logic, v_0 must contain only k elements, since we could simply remove the extraneous elements otherwise. This fact constrains the number of elements in each future vertex, as the length of their leaf-to-cycle path must be less than or equal to k, and, as we've shown, that path must intersect one element from each preceding vertex. Therefore, we are left with the element amounts we initially proposed and the corresponding graph structure.

Now, we want to count the number of elements contained among the k vertices. We know that this amount must be less than or equal to n, so we are left with the inequallity

$$\sum_{i=1}^{k} |v_i| \leq n$$

which we can simplify as follows

$$\sum_{i=1}^{k} k - j + 1 \leq n$$

which evaluates to

$$k^2 - (k-1)k/2 \leq n$$

or

$$k^2/2 + k \leq n$$

Finally, since we know that $k^2/2 + k$ is the smallest number of vertices we need to produce at least one element which requires k operations, we conclude that for n vertices, the maximum number of operations for computing an iteration is $O(\sqrt{n})$. This is far worse than the $O(\log(n))$ we get from the squared-speedup algorithm, but is still hopefully an interesting enough result such that the project was worthwhile.

Real World Performance and Ideas for the Future
After running both algorithms using the "check all points" option built into the program on different random idempotent functions, it appears that the precomputation is faster for the squared solution, but the overall running time is slightly faster for the graph solution.

Our suspicion is that the worst case for the graph algorithm is relatively rare, and generally, the contracted graph generated for a random idempotent results in a something closer to a forest, resulting in a relatively fast computation for most of the domain/codomain elements.

In the future, we could try to find expected time to compute all the points for a random idempotent for each algorithm, in order to capture the intuition behind the real world timings of the program.