Optimal Recomputations using Isomorphisms for Topologically Equivilent problems.

Bryce Summers

October 18, 2014

1 Introduction

Any algorithmic computatation can be expressed in terms of a mapping from input objects to output objects. To convert the objects into output objects a given algorithm will perform a set of sub operations using values from the input to construct values for the output objects. The work the algorithm does can be divided between into two types, the work used to find specific data for computations, and the work of actually performing the computations. In order to acomplish the searching, efficient data structures are needed. Most algorithms can use a custom made data structure to provide constant time searching and therebye enable the work to be dominated by the actual computational operations such as adding integers, comparing numbers, etc. Although there usually exists a data structure that enables constant time searching, it usually takes time to construct such data structures.

In this paper we will discuss a way of optimizing repeated uses of an algorithm by computing the search structures once and then using them for all successive inputs that have the same search topology.

2 Graph Theoretic Viewpoint

If we view our input as a graph $\mathcal{G} = \{V, E\}$, with the vertices V representing the object values and the edges E representing the topology of the input, and we view our output as a graph $\mathcal{G}' = \{V', E'\}$, then the main idea is that we want to construct an ordered sequence of mappings from functions of specific vertices in V and already computed vertices in V' to each vertice in V'. Once we have computed G' , then given a new graph $\mathcal{H}$ with the same topology, the output of the algorithm will be isomorphic to $\mathcal{G}'$. We can therefore compute $\mathcal{H}'$ by iterativly evaluating each of the mapping functions, because every function depends on already previously computed or given values.

Therefore for any algorithm, given a topological equivilance class $\mathcal{E}$, we can compute a G' and a sqeuence of mapping functions such that we can solve any input problem within $\mathcal{E}$ in the time that it takes to evaluate all of the mapping functions.

2.1 Complexity Theoretic Lower Bound

If these mapping functions coorespond to the mathematical acyclic recurrence relations expressed in the mathematics behind the algorithm, then I believe that this computation will be asymptotically optimal, because if the algorithm skipped the evaluation of any part of any of these simple mapping algorithms, then an adversary could provide and input that has a new value that the skipped computation depends on and the algorithm cannot gurantee the correctness of its result.

Therefore the lower bound for these algorithms is equal to the asymptotic complexity of the algorithm we have described, so our algorithm is theoretically optimal.

3 Concrete Examples

In this section we will examine some concrete examples of this algorithmic idea, including the original motivating problem that inspired this paper, which was the recomputation of topologically similar mesh subdivisions using algorithms such as the butterfly algorithm.

3.1 Mesh Subdivision

We can efficiently compute the subdivision of meshes with the same topology that are subdivided using the butterfly subdivision algorithm.

We first compute the subdivided resltant mesh $\mathcal{M}'$ for any mesh $\mathcal{M}$ that is a representative of our chosen topology. While we compute $\mathcal{M}$, we should store which vertices each newly created vertice depends on. We can infer which function is used to derive each of the newly created vertices by the number of vertices that they depend on. For example, if a vertice v depends on 4 old vertices b_1, a_1, a_2, b_2, then we can infer that it is in a boundary case, so the correct derivation of v' is

$$v' = -\frac{1}{16}b_1 + \frac{9}{16}a_1 + \frac{9}{16}a_2 - \frac{1}{16}b_2$$

Likewise, if v' is dependant on 6 inputs, then we can infer that it is the 6 regular case. Please see http://mrl.nyu.edu/ dzorin/papers/zorin1996ism.pdf for more information about butterfly subdivision.

By using our derivation mapping structure, we can elliminate all of the time spent constructing elaborate windged edge data structures if we limit ourselves to input meshes with that have the same topological structure as $\mathcal{M}$, except with different vertice locations. The indices of the vertices should also be the same. Therefore, in applications when we want to deform a simple mesh, we can subdivided the deformed meshes into smooth meshes in linear time, beause each derivation function for butterfly subdivision can be limited to 6 inputs. I have recently used this algorithm to reduce the time it takes to subdivide topologically equivilant meshes multiple frames a second by a factor of 3333 times per second.

3.2 Sorting

An interesting question that I am pondering is whether given an reosonably large amount of precomputation time, a fixed integer n, and polynomial space in terms of n, can I construct a sorting algorithm that sorts inputs of size n in less than $nlogn$ comparisons.

Problems such as sorting do not intuitivly lend themselves to algorithms of the for discussed in this paper, because the majority of their time is spend performing comparison operations. These algorithms spend constant time performing find operations, because they always know exactly which elements they wish to compare next and how to find them due to their implementation using array data structures.

None the less, I have done some thinking about how the time to compute the sorted permutation of an array of n elements efficiently using recomputation. I will now discuss my thoughts as to possible assumptions that we can make about the data.

- No assumptions $\rightarrow nlog(n)$ [Information Theoretic lower bound]
- Permutation is known: $\mathcal{O}(1)$ comparisons, $\mathcal{O}(n)$ time to construct a solution if the permutation is the same, but the numbers differ.
- Every element is within c of its sorted position $\rightarrow \mathcal{O}(n \cdot 2c)$
- The input sequence has a specific sequence of comparisons between its contiguous members $\rightarrow \mathcal{O}(\frac{n}{2}log\frac{n}{2})$ or $\mathcal{O}(1)$ with $\mathcal{O}(\frac{n}{2}!)$ lookup table under some assumptions, which it pretty useless.

3.3 Ray Tracing

We can use this paper's precomputation scheme to efficiently repeated images of scenes using ray tracing assuming that no objects change positions between frames. When raytracing a scene, we can store an ordered set of objects for each pixel that define how the pixel's color was computed. Assuming the recursive depth of reflections is bounded, we can then render images in $\mathcal{O}(pixelnumber)$ time with our number of computations hitting the lower bound on total number of operations necessary to realize the scene.

The key idea is that we store all of the information about a specific scene looked at from a specific camera position. We can then avoid all visibility computations for the remaining frames. Although we cannot ever change the locations of objects, we are free to change all of the properties of the object's materials, colors, reflective amounts, etc. We can also freely change any properties of the lights in the scene as long as we do not change their positions. We can also change the textures for all of the objects, so we can in effect animate texture moving across arbitrary 3d surfaces efficiently.

An example usage for such a scheme would be to efficiently render an animation demonstrating how a scene would look like at various times of day. We could also employ

varous tricks to give the illusion of interesting animation, such as having sequences of lights embedded at locations within the scene that can be turned graudally on and off. We can also hide interesting pieces of the scene in the reflections of scene objects. We can also projcet aribrary 2 dimensional animations onto the various surfaces of our scene that through texture mapping.

4 Musings on Future work

I am thinking about whether their is any insight to be gained in the construction of data structures such as heaps. I am also pondering if their are any assumptions that can be made to benifit algorithms such as sorting.