Algorithm for the Hypergraph Traversal Problem

Masters Thesis: Roscoe Casita
Advisor: Dr. Boyana Norris

June 3, 2017

Abstract

We introduce the hypergraph traversal problem along with the known solutions: Naive, Branch and bound, Polynomial space (recursive). We introduce a polynomial space (serial) version of the algorithm. Odometers are introduced as an implementation detail for future reference. A rigorous test system is used to ensure completeness. Lastly performance is measured in comparison to the other systems.

Contents

1 Introduction

"The theory of hypergraphs is seen to be a very useful tool for the solution of integer optimization problems when the matrix has certain special properties" - C. Berge [1]

2 Definitions

2.1 Definition: List

This paper assumes the reader is familiar with the basic data structures lists, arrays, stacks, queues, trees, graphs etc. Let a list l be an ordered list of things $\{t, i\}$ where each thing t must be the same type, and t is only distinguishable by its index i. The following statements are assumed to be familiar notation and self explanatory:

Algorithm 1 Summorial(N)

```
1:  function SUMMORIAL(n) // N*(N-1)/2 , THIS IS NOT EFFICIENT
2:      l ← ∅ // initialize an empty list, type = list of int
3:      q ← ∅ // list used as a queue, type = list of (list of int)
4:      s ← ∅ // list used as a stack, type = list of int
5:      count ← e.size() // count is now zero, type = int.
6:      for all i ∈ 0, 1, .., n do
7:          l[i] = i // ith index is set to the index value.
8:      l.push(0) // add a zero to the end of the list.
9:      q.push(l) // push the list on the queue.
10:     l.empty() // the list is now empty list, q[0] is not affected.
11:     while q.size() > 0 do // sum over all the lists (just 1 this time)
12:         for all i ∈ q.pop() do // for each element in the sub list
13:             s.push(i) // push the integer
14:     while s.size() > 0 do // sum the list
15:         count ← count + s.pop())
16:     return count // return the summorial of N: N*(N+1)/2
```

2.2 Definition: Odometer [3]

An odometer is an ordered multiset of integer numbers. Let an odometer o be an list of integers n and indexes $\{n, i\}$. The i^{th} indexable integer of an odometer can be written $n_i = o[i]$. Integers n can be repeated, they are distinguished via their index. Indexes i are unique non-repeating whole numbers from $[0, \infty]$. The size of the odometer is written as $o.size()$, is the count of $\{n, i\}$.

An odometer is a construct used extensively throughout this paper as it can be treated as an ordered set of numbers, an unordered bag of numbers, as an instance container to store state. As an unrestricted list of numbers the odometer is similar to a turing machine tape.

Simple common functions such as Union, Intersection, etc are defined in the code Appendix C. Additional short circuit versions of specific test functions implemented for future high performance reasons.

An example is instead of implementing hitting set functions that reason about hyperedges with vertexes, the equivalent function that reasons about odometers is used instead. As bit vectors are implementation dependent standard lists of integers are used instead. Thus M^N values are available, where M is the count of integers, and N is the number of values that integers can take on. Thus the complexity is $O(N^\infty)$ in the general case without restrictions.

2.3 Definition: Unrestricted Hypergraph

The traditional hypergraph definition $H = (V, E)$ is terse for implementers. Traditionally a hypergraph is defined as a collection of sets where there is no ordering and repeated elements are not allowed. The following definitions were used to implement the hypergraph interface. The odometer is of particular interest as it can be used independently from hypergraphs for linear integer optimization techniques.

Let a hyperedge e be a list of vertexes: $e = \{v, i\}$. The i^{th} indexable vertex of e can be written $v_i = e[i]$. Vertexes v can be repeated, they are distinguished via their index. Indexes i are unique non-repeating whole numbers from $[0, \infty]$. The size of the hyperedge written as $e.size()$ is the count of $\{v, i\}$.

Let an unrestricted hypergraph U be a single hyperedge *nodes* and the two functions *OtoE* and *EtoO*. *OtoE* is the surjective function to map a given odometer to a hyperedge. *EtoO* is the injective function to map a given hyperedge to an odometer. The hyperedge *U.nodes* cannot repeat any vertexes v for the function *EtoO* to behave correctly.

Given these definitions, the following is now possible given a hypergraph: A hyperedge can be constructed from an odometer. An odometer can be constructed from a hypergraph. While the functions in the paper use hyperedges the code uses odometers in place of hyperedges. Thus every instance of a hyperedges can be converted to an instance of an odometer, and every instance of an odometer can be converted to an instance of a hyperedge.

Specifically the odometer is an instance of a set of integer numbers that can be reasoned about independently of a hypergraph. The code implements some common set functions that allow the constraints to be expressed, such as union, minus, include short circuit versions of functions for faster performance.

Notice that these functions provide polynomial time access to all permutations, combinations, repeats, patterns etc. Thus reasoning about a hyperedge is equivalent to reasoning about its corresponding odometer, and vice versa. Vertex data can be complex and large, thus reasoning about the odometer in place of the hyperedge is for performance and interesting reasons noted later.

Algorithm 2 OdometerToHyperedge

```
1: function OToE(U, o)
2:     e ← ∅
3:     size ← U.nodes.size()
4:     for all {n, i} ∈ o do
5:         // where -2 % 7 = 5 etc.
6:         e[i] ← U.nodes[n%size] // convert an integer number to index.
7:     return e
```

Algorithm 3 HyperedgeToOdometer

```
1: function EToO(U, e)
2:     o ← ∅
3:     for all {v_e, i_e} ∈ e do // use hashmap of v to i
4:         for all {v_n, i_n} ∈ U.nodes do // to reduce O(n2̂) to O(n)
5:             if v_e = v_n then
6:                 o[i_e] ← i_n // lookup index and save as integer.
7:     return o
```

2.4 Definition: Normal Hypergraph [1]

Let a *normal* hypergraph be $H = (V, E)$ where V is a list of vertexes v, i, E is a list of hyperedges e, i where each hyperedge e is a subset of V. The following trivial restrictions must be imposed to get the expected behavior out of a *normal* hypergraph given the unrestricted list definitions. No hyperedge contains a duplicated vertex. Every vertex in all hyperedges is contained in the hypergraph list of vertexes. There are no duplicate hyperedges. The maximal size of a hyperedge is the size of all hypergraph vertexes. Every vertex exists in at least one hyperedge. There are no duplicate vertexes in the hypergraph.

$$\forall e \in E, \forall v, v' \in e | v \neq v'$$
$$\forall e \in E, \forall v \in e | v \in V$$
$$\forall e \in E, \nexists e' \in E | e = e'$$
$$\forall e \in E | |e| \leq |V|$$
$$\forall v \in V, \exists e \in E | v \in e$$
$$\forall v \in V, \nexists v' \in V | v = v'$$

2.5 Definition: Simple Hypergraph

Let a *simple* hypergraph be $H = (V, E)$ as *normal* hypergraph with the additional restriction that no hyperedge fully contains any other hyperedge.

$$\forall e, e' \in E | e \not\subseteq e' \wedge e' \not\subseteq e$$

2.6 Definition: Transversal of a Hypergraph

Let the transversal of a hypergraph $T \subseteq H.V$ be a minimal hitting set of all the hyperedges of a hypergraph such that $IsHittingSet(T, H.E) = true$.

2.7 Problem: Find all minimal transversals for a given hypergraph

There are $2^{|V|}$ possible combination sets that can be derived from the hypergraph $H = (V, E)$. There are potentially thus $2^{|V|}$ transversals that need to be enumerated. Consider all transversals of the hypergraph

$$H = (\{\{1, 2, 3\}, \{A, B, C\}\}, \{1, 2, 3, A, B, C\})$$

$$T = (\{1, A\}, \{1, B\}, .., \{3, C\})$$

Thus every piecewise combination of this hypergraphs hyperedges is a valid transversal, thus the maximal upper bound on the total number of minimal transversals of all simple hypergraphs is $O(2^{|V|})$ total transversals. Thus scalable algorithms must use polynomial space storage and exponential time to enumerate the potentially exponential number of traversals.

3 Naive solution

4 Branch and Bound

5 Polynomial Space Solution (Recursive)

6 Pseudo-Polynomial Space Solution (Serial)

7 Testing

Hypergraphs are a relatively new data structure; large datasets are not currently modeled as hypergraphs so ensuring correctness is the onus of the implementers. We seek to prove the algorithm is correct for a large set of small hypergraphs and a small set of large hypergraphs. The following algorithm will generate all simple hypergraphs with node count N. The complexity is exponentially exponential on the order of $N!^{N-1!^{N-2!^{\cdots}}}$.

Algorithm 4 Generate All Hypergraphs over Nodes

```
1:  function GenHypergraphs(Nodes,CallbackFunc)
2:      V = Nodes
3:      E = Nodes
4:      H = (V, E) // Single hyperedge of all nodes
5:      CurrentQueue.push(H)
6:      WorkQueue.push(CurrentQueue)
7:      while !WorkQueue.empty() do
8:          CurrentQueue ← WorkQueue.pop()
9:          H ← CurrentQueue.pop()
10:         if !CurrentQueue.empty() then
11:             WorkQueue.push(CurrentQueue)
12:         CallbackFunc(H) // process generated hypergraph
13:         CurrentQueue = GenerateHypergraphChildren(H)
14:         if !CurrentQueue.empty() then
15:             WorkQueue.push(CurrentQueue)
16:
17: function GenerateHypergraphChildren(H
18:     Children ← ∅
19:     for all edge ∈ H.edges do
20:         new_edges ← ∅
21:         for all vertex ∈ edge do
22:             add_edge ← edge \ vertex
23:             if add_edge ⊈ CurrentHypergraph.edges then
24:                 new_edge ← new_edge.push(edge vertex)
25:         AddHypergraph ← Hypergraph(new_edges)
26:         Children ← Children.push(AddHypergraph)
27:     return Children
```

8 Performance

9 Future Directions

The serialized version of the

10 Bibliography

References

[1] C. Berge. *Hypergraphs*. North-Holland Mathematical Library, 1989.

[2] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdag, R. Heaphy, and L. A. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–11. IEEE, 2007.

[3] P. P. Fuchs. *Permutation Odometers*. www.quickperm.org/odometers.php, 2016.

[4] B. Molnár. Applications of hypergraphs in informatics: a survey and opportunities for research. *Ann. Univ. Sci. Budapest. Sect. Comput*, 42:261–282, 2014.