Shantanu Mantri

Massive Graph Analytics

4/19/2019

Final Project Report—Stoer-Wagner Mincut Algorithm

Introduction & Motivation:

The Stoer-Wagner algorithm was proposed in 1995 by Mechthild Stoer and Frank Wagner [3]. The algorithm is a recursive solution to the Minimum Cut Problem, which takes an input Undirected Weighted Graph G = (V, E) and outputs a partition of the vertices into two subsets such that the sum of the weighted edges from one side to the other is of minimal weight.

The algorithm can best be described by the following: we start with a graph G and a randomly chosen vertex s. We then find the minimum cut for the vertices s and some other vertex t. Thus, the {s,t} partition will yield the minimum local value. We then collapse said s-t edge and replace the value with our minimum local value found. We then recurse on the collapsed graph by choosing another random vertex and find the Mincut. We repeat this until there are only two vertices left. We keep a history of the smallest cut, and return said cut. We can also re-create the best partition by keeping a memory of the best cut phase and applying it on the original graph.

The pseudo-code for such an algorithm is described below:

Stoer-Wagner Minimum Cut Algorithm

```
min_cut_value = ∞
contractions = []
for i = 0 … (|V| - 1):
   u = random_element(V)
   A = set([u])
   H = heap()
   For vertex, edge in u.neighbors():
      H.insert(vertex, -edge.weight)
   For j in range(v – i – 2):
      u, q = H.pop()
      If vertex not in A:
         H.insert(vertex, -edge.weight if v not in H else H[v] – e[‘weight’])
   vertex, cut_value = H.min()
   if cut_value < min_cut_value:
      min_cut_value = cut_value
      best_phase = i
//recreate graph looking at best phase
   contractions.append((u,vertex))
//collapse graph
   for w, edge in vertex.neighbors:
      if w != u:
         G.add_edge(u, w, weight=edge.weight)
```

```
        Else:
            G[u][w].weight += edge.weight
//remove node
    G.remove(v)
//code to re-create graph if needed
    ideal_cut = contractions[best_phase]
     Partition = shortest_path(G,ideal_cut)
return min_cut_value, partition
```

Due to the binary heap insertion time being log(n), the total running time of Stoer-Wagner is O(v (e + v) \ log v). This is of course optimized due to a special design change in the heap which will be discussed later in the report.

Approach/Methodologies:

To effectively improve the runtime of Stoer-Wagner, various strategies were employed and tested against other graph frameworks. Some key changes in the implementation include: the setup of the heap, the graph data structure created, and the edge class.

Setup of the Heap:

The heap was slightly altered from the traditional binary heap. It is important to note that the elements in the heap are a tuple of two crucial values: the node id and weight. The main problem was the idea of deleting a node id entry which already existed in the heap. In the general binary heap, time would be spent trying to remove the duplicate element pair (O(log(n))) and then add the new pair (O(log(n))). If we wanted to search the heap, it would add an additional O(n) time.

An easy fix for this is to back the heap with a HashMap of the node and smallest weight. For example, if (node_id = 3, weight = 100) were added to the heap, then the HashMap[3] = 100. Later, if we wanted to add another pair (node_id = 3, weight = 50), we could "invalidate" the previous value by setting HashMap[3] = 50. When we start popping values from the heap and look for the minimum value, we might eventually come across the (3,100) invalid pairing. Since we already know that 50 was the lowest edge value we say corresponding to node_id = 3, we can ignore it and pop the next one instead of choosing it as our "u" (see pseudo-code). This saves time due to searching the heap to see if duplicates exist, and then removing said duplicates (O(nlogn)).

Setup of Graph Data Structures:

The data structure used to store node and edge information was an Edgemap. An Edge class was also created. The Edgemap took up O(v + e) space, and was much more space efficient than an adjacency matrix. Since getting the neighbors of a vertex need to be done quickly in the Stoer-Wagner implementation, the O(1) query time was desirable. The Edgemap was backed by a doubly linked list, since one vertex can have multiple neighbors. This linked list is only important for node removal purposes.

Failed Implementations & Experimentations:

An attempt was also made to parallelize Stoer-Wagner and use a B+ Tree instead of a heap. Unfortunately, neither of these attempts yielded a significant improvement in time, and in most benchmarks increased the time taken to retrieve a Mincut value.

Some possible reasons for the failure to parallelize Stoer-Wagner was the nature of the algorithm itself. The bottleneck of this algorithm essentially lies in the heap, since the removal of nodes and graph collapsing can be done through a single for loop containing O(n) time removals for one phase. On the other hand, the insertion and deletion function for the heap is called multiple times within a single phase, creating the bottleneck. Thus, the best way to improve the Stoer-Wagner running time is to focus on creating a faster heap. After using OpenMP in the main for loop which pops off, inserts, and deletes values in the heap, there was a slight increase in the time taken for graphs having 2^5, 2^6, and 2^7 vertices. This is because a heap insertion or removal action is sequential. Since it is done atomically, OpenMP can't run multiple threads without having to wait on another thread to finish. This defeats the point of parallelism and there is not much scope to improve this section of the code.

The second attempt at parallelization was to improve the removal portion of the algorithm. Since multiple node connections were being deleted in the Edgemap, it was possible to parallelize this portion. Each node has its own unique list that has elements which will be deleted, and repetitions are not allowed. Thus, a pragma omp parallel shared memory call would theoretically improve the system. Benchmarking with graphs having 2^5, 2^6, and 2^10 vertices showed no major improvement in the time taken to succeed. A possible reason for the lack in temporal improvements was the fact that the removal algorithm wasn't time intensive in the first place, and the time needed to create and sync extra threads in OpenMP essentially nullified any potential gains from parallelism. In other words, the number of nodes that needed to be deleted in an individual deletion function never reached a number that would justify a consistent use of parallelism.

The third attempt to improve the algorithm was to change the heap into a B+ tree and examine any changes in running time. Some relevant properties of B+ trees are that duplicate keys can be stored, search times are faster, and inserting an element takes O(log(N)) time. The B+ tree tested with a branching factor of 5, 10, and 15 did not perform significantly better than the heap. Therefore, it didn't make sense to keep the B+ tree implementation, and it was once again replaced by the heap. The other concern with the B+ tree implementation was that the main advantage of B+ trees was never being utilized. The main point of a B+ tree is to get a quick-search and query of a value given a particular key. We don't know the value we are looking for during the Stoer-Wagner cut phase. Therefore, there is no real advantage in keeping a B+ tree over a binary heap.

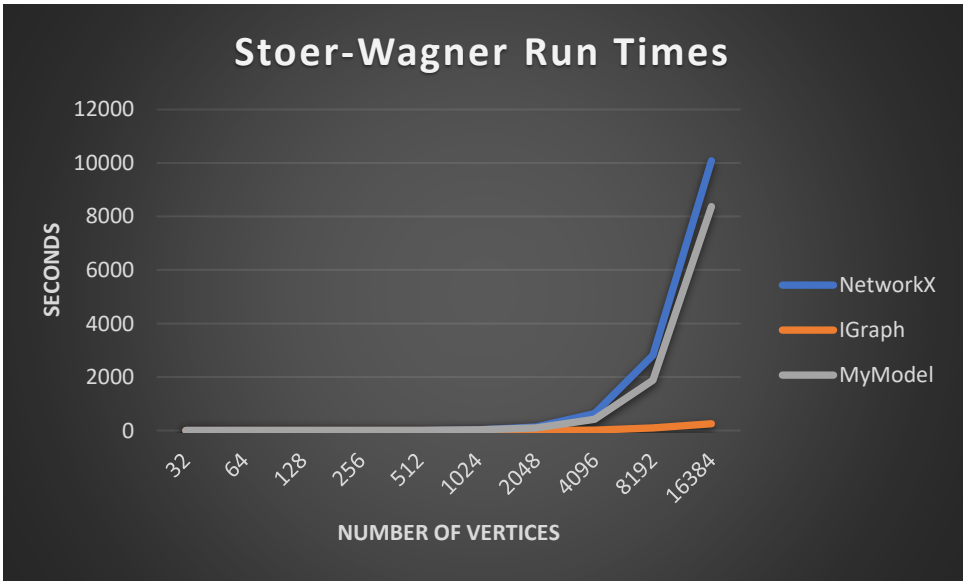
Results:

The CPU used to test the code was an Intel I7 Core Processor on the Lenovo Y700 Ideapad Notebook. This computer also includes an AMD Radeon R9 M375 GPU, but this was not used during the experiments. The graph used for testing was the RMAT Graph generated

with 2^n nodes and 16*n edges through the GTGraph generator. The table below shows the runtimes in seconds for the Stoer-Wagner algorithm.

Vertices	Edges	NetworkX	Igraph	My Model	NX vs IG Speedup	MM vs IG Speedup	NX vs MM Speedup
32	512	0.012896	0.000993	0.01	12.98690836	10.07049345	1.2896
64	1024	0.064947	0.00496	0.047	13.09415323	9.475806452	1.381851064
128	2048	0.293101	0.015873	0.203	18.46538147	12.78901279	1.443847291
256	4096	1.43044	0.053109	1.015	26.93404131	19.11163833	1.409300493
512	8192	6.051208	0.189809	4.671	31.88051146	24.608949	1.295484479
1024	16384	28.192663	0.810944	21.339	34.76524026	26.31377752	1.32118014
2048	32768	128.766456	3.102498	94.65	41.50412216	30.50767478	1.360448558
4096	65536	632.469027	14.39045	418.244	43.9506019	29.06399326	1.512201076
8192	131072	2810.076445	102.0098	1877.901	27.54711968	18.4090236	1.496392219
16384	262144	10083.3604	251.8367	8369.131	40.03928879	33.23237887	1.204827645

As expected, the C++ Stoer-Wagner implementation consistently beat the NetworkX implementation. However, the proposed model was much slower than the Igraph implementation, which was at times almost 40 times as fast as NetworkX, and 20 times as fast as the proposed model. As seen in the graph below, Igraph's implementation combines various optimizations in the data structure and heap to allow fast runtimes.



The breaking point for both the proposed model and NetworkX was 4096 vertices, or 2^12 nodes. An interesting thing to note is that IGraph's Stoer-Wagner algorithm has a theoretical runtime of O(|N||M|+|N|^2 log|N|) [1]. NetworkX and the proposed model have a running time of O(N (M + N) \log N). These should be equivalent, yet IGraph outperforms the proposed model.

Conclusion:

While the proposed model consistently beats the minimum benchmark that is NetworkX, it has trouble performing against Igraph. Some possible reasons for this discrepancy include the data structures used and the optimization of the heap structure. Igraph's data structures are all implemented in C/C++, and they contain very specific optimizations made specifically for parsing and processing graphs. In fact, Igraph has its own "optimized heap" class that it can use for Stoer-Wagner. On the other hand, the proposed model uses Vectors, a Priority Queue, and Maps which are all parts of the standard C++ libraries. This could result in a significant disadvantage, as the data structures are not optimized to handle graph-based structures, adding some lag time. Similarly, one can argue that it was the heap structure that caused the largest lag time. This was proven through testing to identify the "bottleneck" of the algorithm. When running the proposed model on 2^12 vertices, it turned out that the running time was about 420 seconds. About 380 seconds, or approximately 90% of the algorithm was spent on inserting, removing, and popping off the heap. When looking at the pseudo-code, that would be the following portion:

```
For j in range(v – i – 2):
    u, q = H.pop()
    If vertex not in A:
        H.insert(vertex, -edge.weight if v not in H else H[v] – e['weight'])
```

Multiple calls heapify after the insertion function creates the time lag. Since the heap is sequential, it is difficult to effectively parallelize. However, there was a paper published in 1992 called Parallel Heap: An Optimal Parallel Priority Queue [2], which describes an insertion of O(p) objects in O(log(n)) time, where p is the number of available processors. Unfortunately, the code for this heap was unavailable, and therefore unable to be tested in the current iteration. The general idea in this paper was to do the following:

1. **Insertion**: Merge k items to add with the r items already in the node to form a single sorted list. Find the smallest r nodes in the list and place them in the current node. Take the k larger items and process down the child nodes. The child is now the current node in the next iteration.
2. **Deletion**: Each deletion update processes by "levels" in the heap. Additional data structures are used to maintain this. Flags such as insert flags and delete flags are also used.

In general, the heap is maintained through sorting algorithms instead of the traditional heapify algorithm, particularly Cole's Sorting Algorithm. [2] Other variants of sorting algorithms also exist, so it can be assumed that an optimized heap probably builds off of this structure.

Future Work:

To build off the work currently done on improving the Stoer-Wagner algorithm, the next steps would be to integrate it with HornetsNest to test the theory of whether it is the Graph data structure that could be a potential bottleneck for the algorithm. Another important check would be to see how the optimized binary heap is implemented in Igraph, and to try and implement a

similar version in Hornet. The optimized heap could be implemented like the Parallel Priority Queue Paper. Any changes in time taken to solve the algorithm will be attributed to differences in the implementation of the Graph data structures.

References

[1] (n.d.). Retrieved from https://igraph.org/c/doc/igraph-Flows.html

[2] N. Deo and S. Prasad, \Parallel heap: an optimal parallel priority queue". J. Supercomput. 6(1), pp. 87{98, 1992.

[3] Stoer, MECHTHILD, and Frank Wagner. “A Simple Min-Cut Algorithm.” *Https://Fktpm.ru*, ACM, 1997, fktpm.ru/file/204-stoer-wagner-a-simple-min-cut-algorithm.pdf.