An Experimental Evaluation of Two Different Algorithms for Global Minimum Cut

ZHONGWEN ZHANG

University of Western Ontario

zzhan896

zzhan896@uwo.ca

250986857

Introduction

As for the simple minimum cut problem with fixed source and sink, we can use its dual form, that is max flow problem according to "Maxflow-Mincut" Theorem, to solve this problem. However, the global minimum cut problem will be, in a sense, an extension to the "st" minimum cut problem. In the global minimum cut problem, our goal is to find a cut of minimum cost that partitions the graph into two-nonempty sub-graph, where no vertices s or t are specified [1].

In this project, I will implement two algorithms solving global minimum cut problem. One is based on max flow problem while the other one is created by Stoer and Wagner. The report will be organized in such structure as below: 1) analyzing the time complexity of these two algorithms from theoretical perspective 2) experiment method 3) result 4) discussion 5) conclusion.

Theoretical analysis

Here I will name the first algorithm "MaxMincut" and the second one "Stoer_Wagner_mincut". As below, I give the pseudo code for these two algorithms respectively.

Algorithm: MaxMincut (G, s)

Input: G—Graph consisting of |V| vertices and |E| edges; s—start vertex specified manually.

Out: Value of minimum cut and subsets after cut

mincut ← {}

mincutset ← {}

for each v ∈ V, do {

    if v not equals s, then{

        calculate the Maxflow m on the current graph with source-s and sink-v

        find the minimum cut set ms on the residual graph

        add m to mincut

        add ms to mincutset

    }

}

find the minimum value minc in mincut and its corresponding subsets minms from mincutset

return minc, minms

Algorithm Stoer_Wagner_mincut(G,s)

Input: G—Graph consisting of |V| vertices and |E| edges; s—start vertex specified manually.

Output: Value of minimum cut

While |V| > 1, then {

MINIMUMCUTPHASE(G,s)

If the cut-of-the-phase is lighter than the current minimum cut, then {

Store the cut-of-the phase as the current minimum cut

}

}

Return: the current minimum cut

Function: *MINIMUMCUTPHASE(G,s)*

A ←{s}

While A not equals V of G, then {

Add to A the most tightly connected vertex

}

Store the cut-of-the-phase and shrink G by merging the last added vertex into the second last added vertex

Then, I will analyze the time complexity of these two algorithms based on the data structure I used to implement them.

For both algorithms, adjacent matrix is used to represent the graph. For simplicity, I will use AJM as the abbreviation of the adjacent matrix. In addition, since different structured graph will be considered, the different choice of start vertex should not influence the experiment result and I will always set it to 0.

In the MaxMincut algorithm, we will calculate |V|-1 times max flow problem and find the minimum one as the global min-cut. In the implementation of max flow, I used Breadth-First-Search (BFS) to complete searching the augmenting path and used a list to record the path. The theoretical time complexity for the MaxMincut algorithm is $O(|V|^2|E|^2)$.

In the Stoer_Wagner_mincut algorithm, we will calculate |V|-1 times MINIMUMCUTPHASE and choose the minimum one among these phases as the global min-cut. Since the bottleneck of time complexity lies in the way how we select the most tightly connected vertex every time in the

function MINIMUMCUTPHASE, different data structure and different applied method will cause relatively big change in total time complexity, from $O(|V|^4)$ to $O(|V||E| + |V|^2 \log|V|)$ [2].

Generally, we can find that the running time of Stoer_Wagner_mincut algorithm should be better than MaxMincut algorithm theoretically.

Experiment Method

In the stage of experiment, I will mainly design different structured graphs and compare the running time of the two algorithms. Besides, I will use different methods to implement the Stoer_Wagner_mincut algorithm and compare their performances among these different implementations on different structured graph. As below, I will show some types of graphs I have designed:

1) Since the Stoer_Wagner_mincut algorithm only depends on |E| of at most one order while the MaxMincut algorithm depends on $|E|^2$, I designed two different graphs, where one is called "sparse graph" as shown in the Fig 1 and the other one is called "dense graph". Here, dense graph represents a bunch of graphs whose vertices connect to one another and |E| will be consequently calculated as $\frac{|V|(|V|-1)}{2}$. The pseudo code for generating the dense graph is shown as below. In this way, the ratio of |E| over |V| in the two-different structured graph will be the maximum and we can easily observe the relative performance of the two algorithms.

```
Algorithm: GenerateDenseGraph(v)
Input: v—vertex list of the graph
Ouput: linked edge list
edges ← {}
for each i ∈ range of 1 to |V|, do{
    if i not equals the largest vertex index, then{
        for each j ∈ range from i to largest vertex index, do{
            linked the edge of v[i] to v[j] and add it to the edges
            set the edge weight equal to j
        }
    }
}
return edges
```

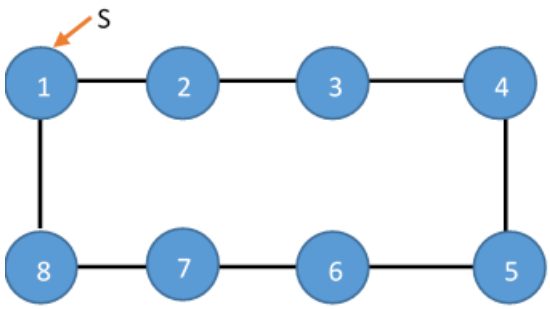

Fig 1. Illustration of the "sparse graph" with eight vertices and eight edges for example. "S" points to the source.

2) I also generated some "medium dense" graphs by the algorithm shown as below:

Algorithm: GenerateMedDenseGraph(v,iter)
Input: v—vertex list of the graph; iter—iteration time
Output: linked edge list
i ← 0
edges ← {}
while i not equals the iter, then{
    shuffle the v
    for j from 1 to |V|-1, then{
        link the edge of v[j] to v[j+1] and add it to edges
        set the edge weight equal to j
    }
return edges

In the detailed implementation, I used the dictionary in Python to represent linked edge list. Since we can not guarantee that in every iteration we will generate totally different linked edges, the number of edges will thus be at most (|V|-1|) * iter.
In this way, we could control the number of edges in the graph to a medium level and the randomness in generating the graph will make the experimental conclusion more general.

Results

Here, I will present some results of experiments based on outlines described as below:

(a) Calculating the running time of MaxMincut and Stoer_Wagner_mincut algorithm respectively on three different structured graphs with vertex number ranging from 5 to 60. In this experiment, I will use the naïve implementation of Stoer_Wagner_mincut of which time complexity is $O(|V|^4)$. In the "medium graph", I set the ratio of |E| over |V| as approximately 10.
(b) Calculating the running time ratio of MaxMincut over Stoer_Wagner_mincut on three different structured graphs with vertex number ranging from 5 to 60.

(c) Picking up a reasonable vertex number, say, 200, and comparing the running time of different implementation of Stoer_Wagner_mincut.

A. Results on experiment (a)

In order to find out how different vertex number and different structure of graph will, in practice, influence the running time of MaxMincut and Stoer_Wagner_mincut algorithm respectively, I had set the vertex number ranging from 5 to 60 and tested it on three types of graphs. I picked up 60 as the upper limit for the vertex number range because I found that setting at this time range would both show valuable experiment results and keep the program from running for too long.

We can observe from the Fig 2 as below that the Stoer_Wagner_mincut has almost the same performance on three different types of graph and running time increases with the growth of vertex number. On the contrary, the MaxMincut shows obviously different performance on "medium graph" while performs fairly well on the other two types of graph.

In the theoretical analysis above, we can see that the time complexity of MaxMincut algorithm depends on the second order of |E|. However, the experiment results present that the maximum ratio of |E| over |V| does not increase the running time strikingly while on the "medium graph" the running time shows rapid growth. This kind of contradictory with theory analysis is related to detailed implementation of the algorithm and graph structure and this will be further discussed in the next section. On the other hand, the theoretical analysis of Stoer_Wagner_mincut time complexity which is $O(|V|^4)$ matches the experimental results well. In general, the graph structure does not influence the performance of Stoer_Wagner_mincut algorithm.

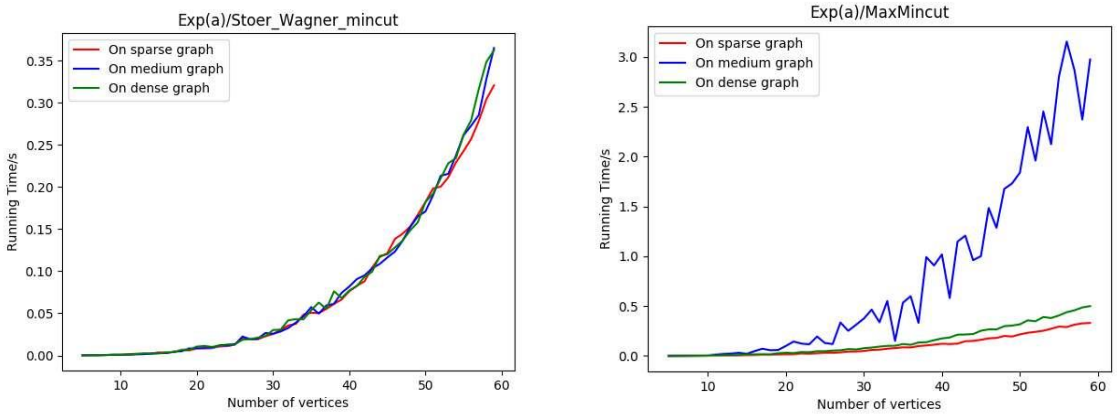

Fig 2. Running time of Stoer_Wagner_mincut and MaxMincut algorithm on three different types of graph with growth of vertex number.

B. Results on experiment (b)

In order to provide more intuitive result on the comparison between these two algorithms, I calculated the ratio of the running time as shown in Fig 3. We can easily observe that for smaller number of vertices the Stoer_Wagner_mincut performs better on every type of graph.

However, with the growth of vertex number, the MaxMincut gradually performs better specifically on “sparse graph” and “dense graph”. The reason is similar to that of the contradictory result as above and mainly because of the specific graph structure which gives “short-cuts” to the implementation of the MaxMincut algorithm.

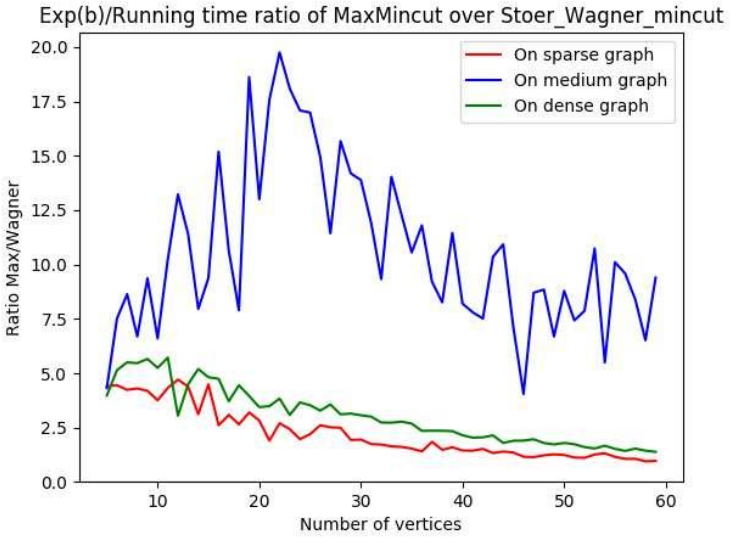

Fig 3. The running time ratio of MaxMincut over Stoer_Wagner_mincut on three types of graph with growth of vertex number. The ratio is much larger on the “medium graph” comparing with the other two types of graph.

C. Results on experiment (c)

In this part, I will set vertex number as 200 which is large enough to reduce the influence of the randomness in practical running time. In the first version of implementation, I just directly searched all vertices in the graph and determine if it belongs to A. If the vertex does not belong to A, then I searched all the vertices in A to calculate the sum of weights. This will cause the time complexity of the Stoer_Wagner_mincut to reach $O(|V|^4)$. Even though $O(|V|^4)$ is still much better than $O(|V|^2|E|^2)$, we can still improve it a lot.

In the first improvement, I tried to copy a vertex list every time calling the MINIMUMCUTPHASE function. When adding a vertex to A, I correspondingly removed that vertex from the copied vertex list. This will help prevent searching the whole vertex set every time and also save the time spent on checking whether the vertex belongs to A. As we can see from the Table 1, the result indeed gets improved, but the improvement is very little.

In the second improvement, I tried not to calculate the sum of weights every time from the scratch, but instead I used a weight-list dictionary to save the sum of weights of every vertex. Every time I removed a vertex and added it into A, I could just update the weight-list dictionary by adding the weight to the corresponding vertices. The idea for this improvement came up from the original paper by Stoer and Wagner [2] and the running time had got strikingly improved to only around 1.5 seconds as shown in the Table 1.

	Naïve version	First improvement	Second improvement
On sparse graph	40.854s	37.553s	1.380s
On medium graph	41.215s	37.695s	1.450s
On dense graph	41.088s	39.924s	1.522s

Table 1. Running time comparison among different implementation of the Stoer_Wagner_mincut algorithm on three different types of graph. The vertex number is all set to 200.

Discussion

In this report, we have shown two kinds of classical algorithms dealing with global minimum cut problem and evaluated their performances on three different structured graphs. Generally, the Stoer_Wagner_mincut performs better except on the "sparse graph" and "dense graph" with large number of vertices. After checking the method applied to generate these two types of graph and the detailed implementation, I found the problem is mainly caused by two aspects: 1. the special structure of the "sparse graph" 2. the weight assigned to edges in the "dense graph". In the following, I will discuss these two problems respectively.

As shown in the Fig 1, we calculated the max flow with fixed source vertex 0 for |V|-1 different sink vertex in each phase. Obviously, there are totally two augmenting paths where one is from source to sink clockwise while the other one is counter clockwise. Therefore, the time complexity for finding the augmenting path should be at most $O(3|V|^2) = O(|V|^2)$ according to the implementation and the total time complexity for the MaxMincut algorithm should be $O(|V|^3)$. Comparing with the naïve implementation of Stoer_Wagner_mincut algorithm of which time complexity is $O(|V|^4)$, the experimental result that MaxMincut becomes faster on the "sparse graph" with growth of vertex number could now make sense. In addition, the weight assigned to the edge will not influence the result since the number of augmenting paths is only two and these two paths are, in a sense, "one-off" paths.

As for the evaluation on the "dense graph", I modified the method to generate the graph by changing the edge weight to a random number between a prefixed range and the algorithm for generating the new "dense graph" is shown as below.

Algorithm: GenerateNewDenseGraph(v,num)
Input: v—vertex list of the graph; num—upper limit of the range to generate the random integer
Ouput: linked edge list
edges ← {}
for each i ∈ range of 1 to |V|, do{
    if i not equals the largest vertex index, then{

for each j ∈ range from i to largest vertex index, do{
randomly generate an integer w from range of one to num
linked the edge of v[i] to v[j] and add it to the edges
set the edge weight equal to w
}
}
}
return edges

Then, I compared the running time of the MaxMincut algorithm on different types of "dense graph". The experimental result is shown in the Fig 4 . The number of iteration times stands for the experimental times and I set it to 7. As for the first two graphs, I set weight to one for every edge in the graph. Then I applied the GenerateNewDenseGraph algorithm to construct graphs for the second and third experiment and we can see that the running time presents sharply increasing while for the last three experiments I used the GenerateDenseGraph algorithm to build the graph and the running time changed less again. In the detailed implementation of MaxMincut, I always searched from the first vertex index when finding the augmenting path. Meanwhile, the GenerateDenseGraph algorithm sets the edge weights growing with the vertex index. By this means, the bottleneck of every searched augmenting path will appear at vertex with small index and, thus, it will reduce the difficulty in searching the augmenting paths.

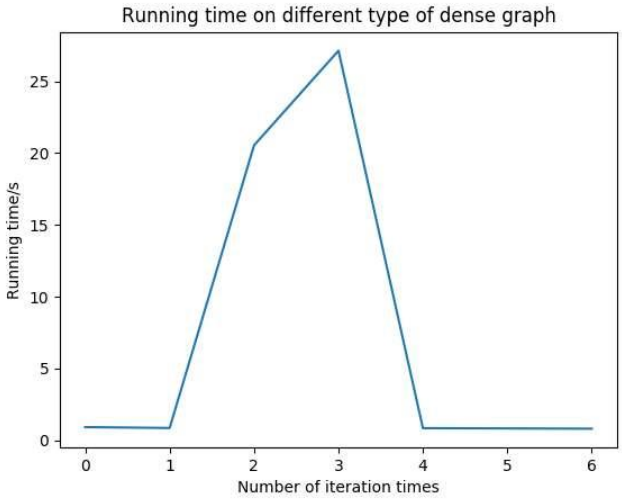

Fig 4. Running time on different "dense graph". Seven examples in total:0,1—setting edge weight to 1; 2,3—GenerateNewDensegraph algorithm; 4,5,6—GenerateDensegraph algorithm

Since I used the naïve version implementation of Stoer-Wagner_mincut algorithm for the experiment (a) and experiment (b), then I reimplemented the Stoer_Wagner_mincut algorithm and plotted the running time ratio curve on "sparse graph" and "dense graph" again, where the "dense graph" was still generated by GenerateDenseGraph algorithm. We can see from the Fig 5 that the Stoer_Wagner_mincut will perform much better than the MaxMincut on both specific types of

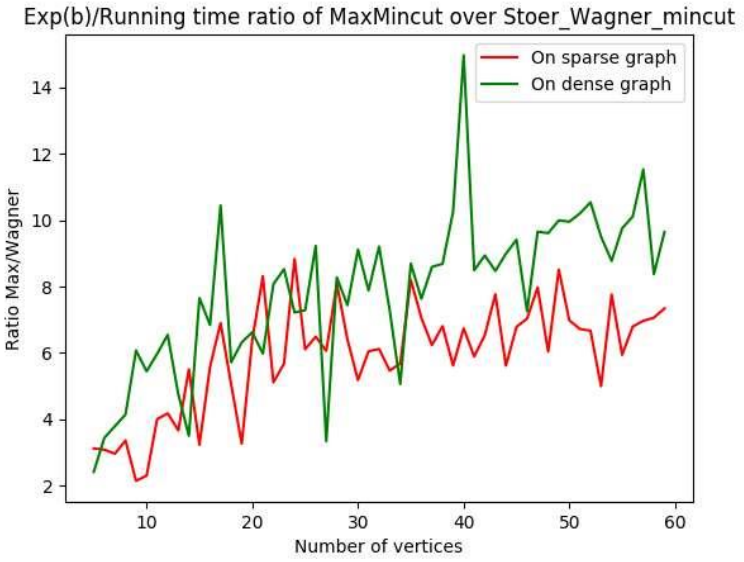

Fig 5. Running time ratio curve. Using GenerateDenseGraph to construct the "dense graph" and applied the Stoer_Wagner_mincut algorithm of the fastest version.

graph with growth of vertex number. On the other hand, the bottleneck of MaxMincut mainly lies in the implementation of max flow problem and, however, this is out of scope of this report.

Conclusion

In this report, I have evaluated the two selected algorithms' performance on serial sets of different graphs. The improved implementation of Stoer_Wagner_mincut algorithm could get much better performance on each type of graph. On the other hand, the MaxMincut algorithm could perform well on the specified "sparse graph" and "dense graph". The graph structure, whether it is dense, sparse or medium, and the weight assigned to each edge will both influence the MaxMincut algorithm' performance regarding the running time.

Generally, the proper implementation of the Stoer_Wagner_mincut algorithm will be our better choice if we intend to deal with any global minimum cut problem within shorter time.

Reference

1. From http://www.csd.uwo.ca/Courses/CS4445a/projects.html.
2. Mechtild Stoer, Frank Wagner. A Simple Min Cut Algorithm. Journal of the ACM, Vol 44, No. 4, July 1997, pp. 585-591.