Advanced Algorithms

Assignment 3: Minimum cut

June 16, 2022

Budai Matteo	2057217
Burke Jamie	2044062
Tlepin Sanjar	2041606

Contents

1 Introduction 2

2 Stoer and Wagner's Deterministic Algorithm 3
- 2.1 Data Structure 3
  - 2.1.1 Graph 3
  - 2.1.2 Node 4
  - 2.1.3 MaxHeap 4
- 2.2 Implementation 4
  - 2.2.1 stMinCut 5
  - 2.2.2 Contract 5
  - 2.2.3 GlobalMinCut 6
- 2.3 Complexity 6

3 Karger and Stein's Randomized Algorithm 8
- 3.1 Data Structure 9
- 3.2 Implementation 9
- 3.3 Complexity 10

4 Results 12
- 4.1 Table with Min-Cut results 12
- 4.2 Graphs of the Time Cost of the two Algorithms 13
- 4.3 Graph of the Time Cost compared to the Discovery Time of Karger and Stein Algorithm 15
- 4.4 Graph of the Time Cost compared to the Asymptotic Complexity of the two Algorithms 16

5 Conclusion 18

1 Introduction

For this assignment, we implemented and analyzed the performance of two algorithms for the min-cut problem for weighted graphs. The algorithms implemented are:

1. Stoer and Wagner's Deterministic Algorithm;
2. Karger and Stein's Randomized Algorithm.

2 Stoer and Wagner's Deterministic Algorithm

General Structure:

```
function GlobalMinCut (G)
    if V = {a,b} then
        return ({a},{b})
    else
        (C1,s,t) ← stMinCut(G)
        C2 ← GlobalMinCut(G/{s,t})
        if w(C1) ≤ w(C2) then
            return C1
        else
            return C2
```

stMinCut with max heap:

```
function stMinCut (G = (V,E,w)):
    Q ← ∅                            // Q max heap priority queue
    for all node u ∈ V do
        key[u] ← 0
        Insert(Q,u,key[u])
    s,t ← null
    while Q ≠ ∅ do
        u ← extractMax(Q)
        s ← t
        t ← u
        for all v adjacent to u do
            if v ∈ Q then
                key[v] ← key[v] + w(u,v)
                IncreaseKey(Q,v,key[v])
    return (V − {t},{t}),s,t
```

2.1 Data Structure

A max heap data structure is used in order to implement Stoer and Wagner's algorithm with a priority queue. There are two classes, Graph and Node, which define the data parameters that are necessary for the MaxHeap class, as well as data parameters that are used in the implementation functions. The Graph, Node, and MaxHeap classes are defined as follows:

2.1.1 Graph

The Graph class takes the graph .txt file as an input, and initializes variables that construct the graph and support the max heap data structure.

- **Initialize**: calls Python's defaultdict dictionary type
- **createNodes**: takes number of nodes as an input, and initializes each node in the node dictionary by calling the Node class.

- **buildGraph**: takes the graph .txt file as an input, and passes the number of nodes to the createNodes function. Futhermore, it passes each connecting node and their edge weight to the **makeNodes** function, which appends to the nodes adjacencyList.

2.1.2 Node

The Node class initializes eight instance variables for each node of the graph

- tag: integer identifier of a node
- key: default key value of null
- parent: default parent value of null
- isParent: boolean value to determine if a node is in a heap, default of true
- index: index of node of the min heap array
- adjacencyList: adjacency list of the node, default is an empty list
- notMerged: boolean value to determine if a node has been merged, used in the contract function
- flagupdate: value to determine if a nodes adjacent cost needs to be udpated, using in the contract function

2.1.3 MaxHeap

The MaxHeap class creates the max heap data structure with an array heap, and is initialized by passing the node dictionary values and the starting node integer tag. In addition to functions that return the standard array heap rules parent, leftchild, and rightchild, the following functions are defined:

- **maxHeapify**: this method is passed an index and checks if any node swaps are required to maintain the max heap data structure, and if so it recursively calls itself until the max heap data structure is achieved
- **shiftUp**: this method is passed an index and properly positions the index element in the array with respect to it's parent in order to maintain the max heap data structure
- **extractMax**: the method extracts the max, or root value, of the array heap. After extracting, it calls the maxHeapify method in order to maintain the max heap data structure

2.2 Implementation

The algorithm is implemented using three functions: stMinCut, Contract, and GlobalMinCut. The GlobalMinCut function is the main function that will return the minimum cut of a graph, and within GlobalMinCut we call stMinCut and Contract. Before calling any of these functions, we must initialize the graph object using the Graph class, and then call the buildGraph function. As we will want to measure the execution time of the algorithm, we start a timer before calling the GlobalMinCut function, and stop the timer once the function returns the minimum cut. The three functions used in the implementation are described as follows:

2.2.1 stMinCut

The stMinCut function is passed a graph and returns two vertices s and t, such that s ∈ S and t ∈ T, and also returns the weight w(S,T), where w(S,T) is the smallest possible among all s,t cuts. We chose to have the stMinCut function return the weight of the s,t minimum cut rather than building a separate function. Using a max heap priority queue, the stMinCut function is implemented as follows:

1. Define an arbitrary starting node, a
2. For each node in the graph, define the key as 0, and define the isPresent value as True
3. Initialize the max heap data structure by calling the MaxHeap class, passing the nodes from the graph and the starting node a. If a is not already the root node, the call to initialize the max heap data structure will re-set the root node as the passed starting node, and will update the index for all other nodes.
4. Initialize s and t as null
5. Now that the MaxHeap object has been created, we will perform the following iterative process until the heap size is zero:
   - Extract the maximum from the max heap data structure by calling extractMax(), which returns the node u with the maximized weight that should be visited next.
   - Set s equal to t, and t equal to u
   - For each node, v, in the adjacency list of u, check if it is present in the array heap. If it is, increase it's key by the weight between u and v, and shift up v in the max heap. Call maxHeapify with the original starting node a in order to preserve the max heap data structure.

   On the last iteration of step 5, s will be defined as the second to last node existing in the max heap, and t will be defined as the last existing node.
6. Initialize the ST_cut_cost and set equal to 0
7. Set ST_cut_cost equal to the sum of all edge weights adjacent to t
8. Return s, t, and ST_cut_cost

2.2.2 Contract

The contract function is passed a graph and two nodes s and t, and returns the graph with s and t contracted into one node:

1. Determine which node between s and t has the minimum tag, and choose this tag to represent the newly merged s,t node
2. Traverse through each node u in the graph and check if it connected to s, t, or both.

- If a node only connected to either s or t, keep that original connecting weight as the weight connecting u to the merged s,t node
- If a node u connected to both s and t, calculate the weight connecting u to the merged s,t node as the sum of the original two connecting weights, w(u,s) + w(u,t)

3. Update the merged s,t node adjacency list to include any changes made in the prior step
4. Remove from the graph the node associated with the tag that was not chosen to represent the merged s,t node and then return the graph

2.2.3 GlobalMinCut

The function GlobalMinCut takes a graph G and returns it's minimum cut cost. It is a recursive function that is performed as follows:

1. Check if the graph only contains two nodes, and if so return the cost of their connecting edge. If the graph contains more than two nodes, continue to the next step
2. Call the stMinCut function passing the graph G, and store the returned nodes s, t, and the s,t minimum cut ST_cut_cost
3. Call the contract function passing the graph G and nodes s and t, and store the returned graph new_G that has merged nodes s and t
4. Recursively call the GlobalMinCut passing the merged graph new_G, which returns the minimum s,t cut from either G and new_G. The final return will be the minimum s,t cut that was found throughout the recursive process, and therefore the global minimum cut.

2.3 Complexity

To calculate the total complexity, we must consider the following components where n in the number of nodes and m is the number of edges:

- stMinCut using a max heap
  - Initialization of each node: O(n)
  - extractMax has complexity of O(log n) and is performed n times in the while loop, so total complexity of O(n log n)
  - The for loop is called O(n) times, the check within the for loop is of complexity O(1), and the shiftUp method is of complexity O(log n). Therefore, the total cost of the for loop is O(m log n).

  Adding these components simplifies to a total complexity of O(m log n).
- GlobalMinCut will call stMinCut n times

Therefore, **the total complexity of the algorithm is O(mn log n)**. We were able to achieve this complexity in our code as seen in the asymptotic complexity comparison shown in the Results section.

3 Karger and Stein's Randomized Algorithm

```
KARGER (G,k):
 min = +∞
 for i = 1 to k:
     t = RECURSIVE_CONTRACT(G)
         if t < min:
             min = t
 return min

RECURSIVE_CONTRACT(G=(D,W)):
 n= number of vertices in G
 if n<=6:
     Gp= CONTRACT(G,2)
     return weight of the only edge (u,v) in Gp
 t = n/√2+1
 for i = 1 to 2:
     Gi = CONTRACT(G,t)
     wi = RECURSIVE_CONTRACT(Gi)
 return min(w1,w2)

CONTRACT(G=(D,W),k):
 n= number of vertices in G
 for i = 1 to n-k:
     (u,v) = EDGE_SELECT(D,W)
     CONTRACT_EDGE(u,v)
 return D,W

CONTRACT_EDGE(u,v):
D[u] = D[u]+D[v]-2W[u,v]
D[v] = 0
W[u,v] = W[v,u] = 0
 for each vertex w ∈ V: except u and v:
     W[u,v] = W[u,w] + W[v,w]
     W[w,u] = W[w,u] + W[w,v]
     W[v,w] = W[w,v] = 0

EDGE_SELECT(D,W)
 1. Choose u with probability proportional to D[u]
 2. Once u is fixed, choose v with probability proportional to W[u,v]
 3. return the edge (u,v)
```

This is a randomized algorithm for the computation of a graph. In the next subsections we explain how we have implemented the data structure and the functions of the algorithm.

3.1 Data Structure

To implement the data structure required for our algorithm, we first read the input file and define the number of nodes and number of edges. After we define these two parameters, we then build the info array with the inputs (node1 , node2, and weight), which is then used to build the V, D, and W fields that are described below.

- **k**: is a constant ($log^2 n$) used by Karger to repeat Recursive-Contract k times and to obtain an error with probability less or equal to 1/n where n is the number of vertices;
- **V**: is the list of nodes;
- **D**: is the list of the sum of the weights of each node;
- **W**: is the list of the graph with 3 parameters(node,node,weight). Each node is connected with the others and if they are not connected the third parameter is set to 0 otherwise is set to the correct weight.

3.2 Implementation

For the implementation of the algorithm we used these functions:

- **Karger(G,k)**:
  1. This is the main function of the algorithm where we set the timeout to 120 seconds to limit the execution time of large instances;
  2. We start the time and we set the minimum cut to infinite;
  3. We iterate k times to obtain an error with probability less or equal to 1/n;
  4. If the time minus the starting time is greater than the timeout we break;
  5. We execute a copy of V,W and D and then we call the function Recursive-Contract;
  6. If we found a value less than our minimum we update it and we set the discovery time;
  7. In the end we print the Minimum Cut, the Total time and the discovery time.
- **Recursive-Contract(V, W, D)**: In this function we follow exactly the function above with our data structure;
- **Contract(s, V, W, D)**: Also in this function we follow the function above. The only difference is that when we select and contract an edge (u,v) then we remove from V the vertex v to respect the contraction;
- **Contract-Edge(u,v, W, D)**: Also in this case we follow the function above (We update D and W with the new values to execute the contraction of the edge) with our data structure;
- **Edge-Select(V1, D, W)**: Edge selecting randome select first node and look for connected other node, algorithm use:
  1. **Random-Select(C)**:

(a) Build cumulative weights vector by input array of weights;

(b) Set random value **r** in range (0, max value of weight);

(c) Run binary search to return node related to selected edge;

2. **binarySearch(array, x)**:

   (a) Special case of binary search according to inequality C[i - 1] $\leq$ r $<$ C[i];

   (b) Divide array in parts;

   (c) Check first value of right part, if random value higher, then go right;

   (d) If value lower check that previous element lower or equal;

   (e) Return related name of node.

3.3 Complexity

Full time complexity of Karger and Stein's Randomized Algorithm is $\sim$ O (n^2 $\log^3$ n). It include counting of complexity of Recursive part properties and Edge selecting part.

4 Results

4.1 Table with Min-Cut results

	Karger Stein			Stoer Wagner	
File	Minimum Cut	Execution Time	Discovery Time	Minimum Cut	Execution Time
input_random_01_10.txt	3056	0.000879765	0.00025177	3056	0.00017786
input_random_02_10.txt	223	0.000859261	0.00022912	223	0.000152111
input_random_03_10.txt	2302	0.000858068	0.000437021	2302	0.000157833
input_random_04_10.txt	4974	0.000911713	0.000257969	4974	0.000146866
input_random_05_20.txt	1526	0.008926868	0.001991749	1526	0.000664234
input_random_06_20.txt	1684	0.009434938	0.001049995	1684	0.000605345
input_random_07_20.txt	522	0.009203196	0.003240347	936	0.000658035
input_random_08_20.txt	2866	0.008773804	0.000980854	3210	0.000651121
input_random_09_40.txt	2137	0.062165976	0.030718088	2137	0.002675056
input_random_10_40.txt	1446	0.046857834	0.010510921	1446	0.002933979
input_random_11_40.txt	648	0.047665834	0.005800962	648	0.002755642
input_random_12_40.txt	2486	0.04701519	0.016011	2486	0.002772093
input_random_13_60.txt	1282	0.230630159	0.029880047	1282	0.007097006
input_random_14_60.txt	299	0.234168053	0.038658857	299	0.007128716
input_random_15_60.txt	2113	0.234682083	0.065416813	2113	0.00684309
input_random_16_60.txt	159	0.232192039	0.028501034	159	0.006592035
input_random_17_80.txt	969	0.48109889	0.062489986	969	0.011644125
input_random_18_80.txt	1756	0.482280016	0.063866138	1756	0.012774706
input_random_19_80.txt	714	0.483850956	0.060369968	714	0.0126791
input_random_20_80.txt	2610	0.493764877	0.06318903	2610	0.012727976
input_random_21_100.txt	341	0.853721142	0.106700182	341	0.020539999
input_random_22_100.txt	890	0.863779783	0.221526146	890	0.019736767
input_random_23_100.txt	772	0.862797022	0.108649969	772	0.019026279
input_random_24_100.txt	2512	4.872792006	1.786714077	1561	0.020540714
input_random_25_100.txt	951	4.011253119	1.302159071	951	0.0440979
input_random_26_150.txt	424	4.024105072	0.33183527	424	0.051584005
input_random_27_150.txt	1153	4.017089128	0.322326183	1153	0.042558432
input_random_28_150.txt	707	4.03592205	0.166624069	707	0.051145315
input_random_29_200.txt	484	8.905862093	0.343484879	484	0.084405184
input_random_30_200.txt	850	8.901889801	0.702427864	850	0.084483862
input_random_31_200.txt	1382	8.931558132	2.484754324	1382	0.082328081
input_random_32_200.txt	1102	8.914821148	0.700302124	1102	0.089802027
input_random_33_250.txt	346	17.71111083	2.714507103	346	0.134936094
input_random_34_250.txt	381	16.99969697	0.669975758	381	0.129019976
input_random_35_250.txt	129	17.21377277	0.667495966	129	0.14840889
input_random_36_250.txt	670	17.08445001	0.659725666	670	0.132274151
input_random_37_300.txt	1137	30.46328282	7.068966866	1137	0.204955101
input_random_38_300.txt	869	30.25916505	1.134328842	869	0.198097944
input_random_39_300.txt	868	43.66593575	3.4303689	868	0.212153196
input_random_40_300.txt	1148	43.96793175	5.732435942	1148	0.213391304
input_random_41_350.txt	676	77.57277179	3.816658974	676	0.286592007
input_random_42_350.txt	290	77.85609221	1.837364197	290	0.286658049
input_random_43_350.txt	818	77.93455815	12.00841498	818	0.288505077
input_random_44_350.txt	175	77.97997689	1.822113037	434	0.283100128
input_random_45_400.txt	508	121.952219	3.014369965	508	0.401754856
input_random_46_400.txt	904	120.510422	18.39590216	904	0.36919713
input_random_47_400.txt	362	122.5628853	21.26115632	362	0.354384184
input_random_48_400.txt	509	123.3320611	2.752584934	509	0.381939173
input_random_49_450.txt	400	120.2518651	12.39884496	400	0.484717131
input_random_50_450.txt	364	121.216306	4.213177919	364	0.456595898
input_random_51_450.txt	336	124.7754698	11.98931384	336	0.449820042
input_random_52_450.txt	639	124.875001	35.25984502	639	0.475036144
input_random_53_500.txt	43	120.5694153	5.117500305	43	0.640682936
input_random_54_500.txt	805	123.3800392	41.22685623	805	0.599419117
input_random_55_500.txt	363	123.5930979	22.01072884	363	0.550251961
input_random_56_500.txt	584	310.762332	5.182160854	584	0.576139927

Figure 1: Table of results

4.2 Graphs of the Time Cost of the two Algorithms

Figure 2: Stoer and Wagner's execution time

Figure 3: Karger and Stein's execution time

4.3 Graph of the Time Cost compared to the Discovery Time of Karger and Stein Algorithm

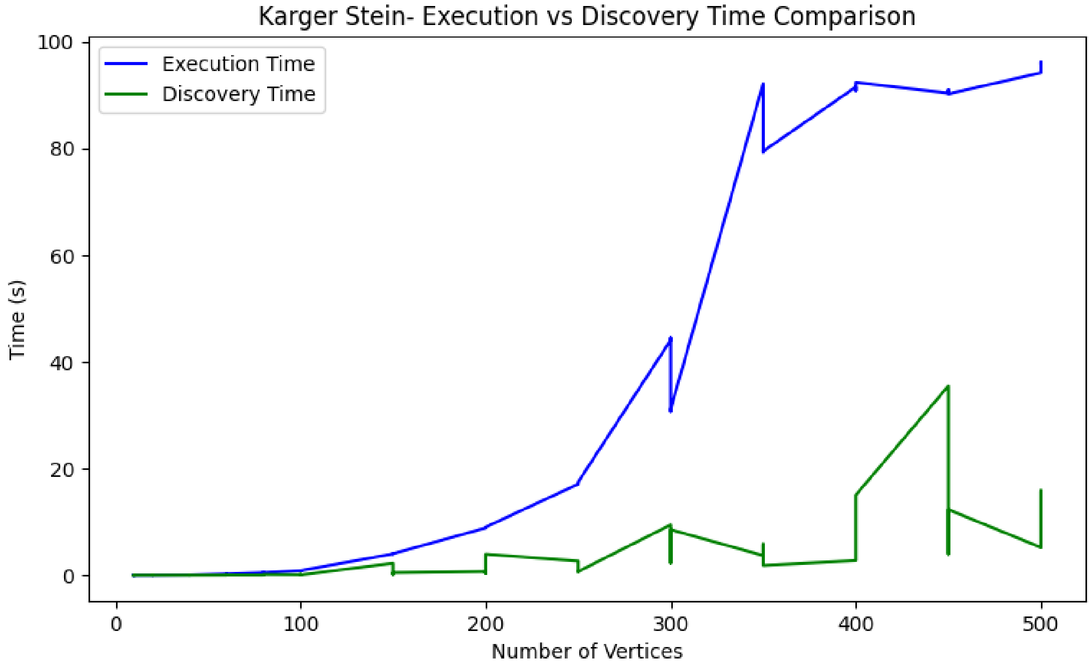

Figure 4: Karger and Stein's Time Cost compared to the Discovery Time

4.4 Graph of the Time Cost compared to the Asymptotic Complexity of the two Algorithms

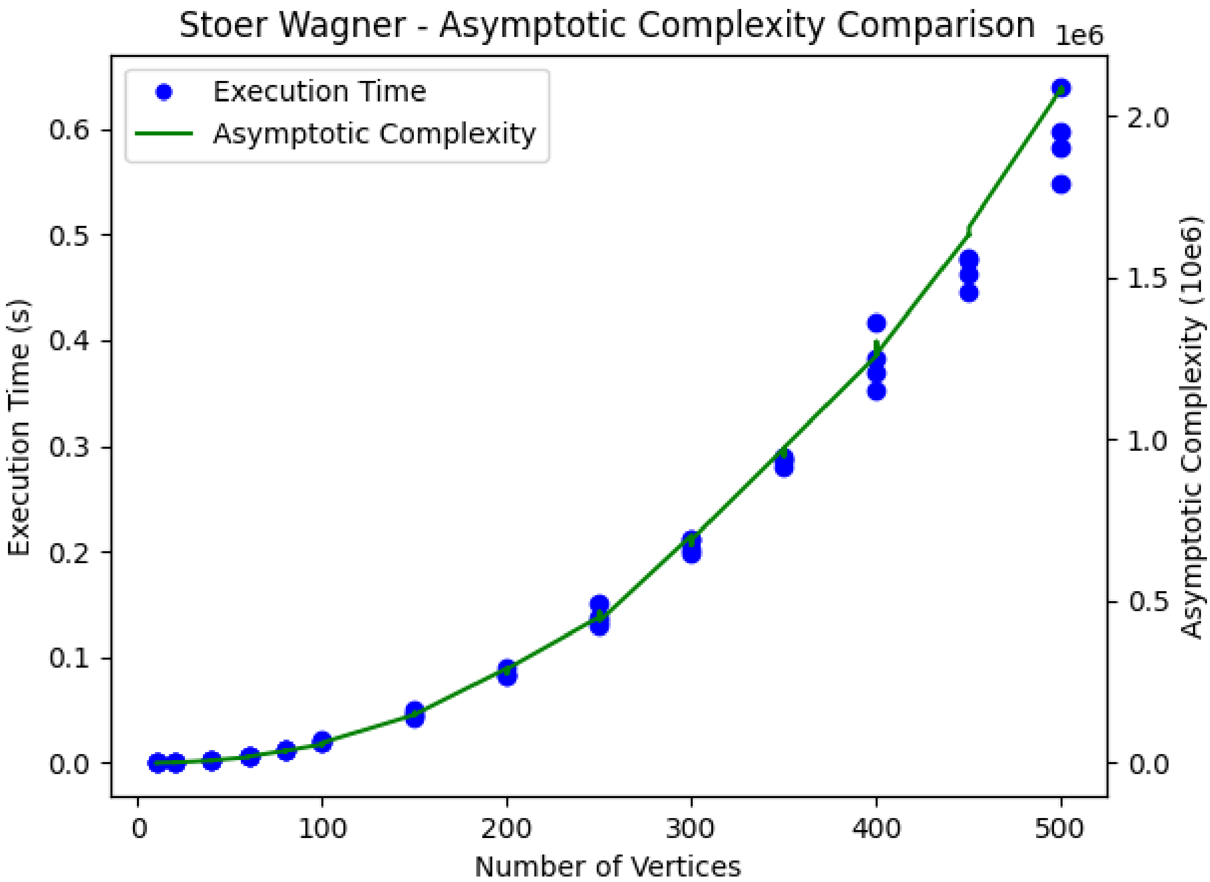

Figure 5: Stoer and Wagner's Time Cost compared to the Asymptotic Complexity

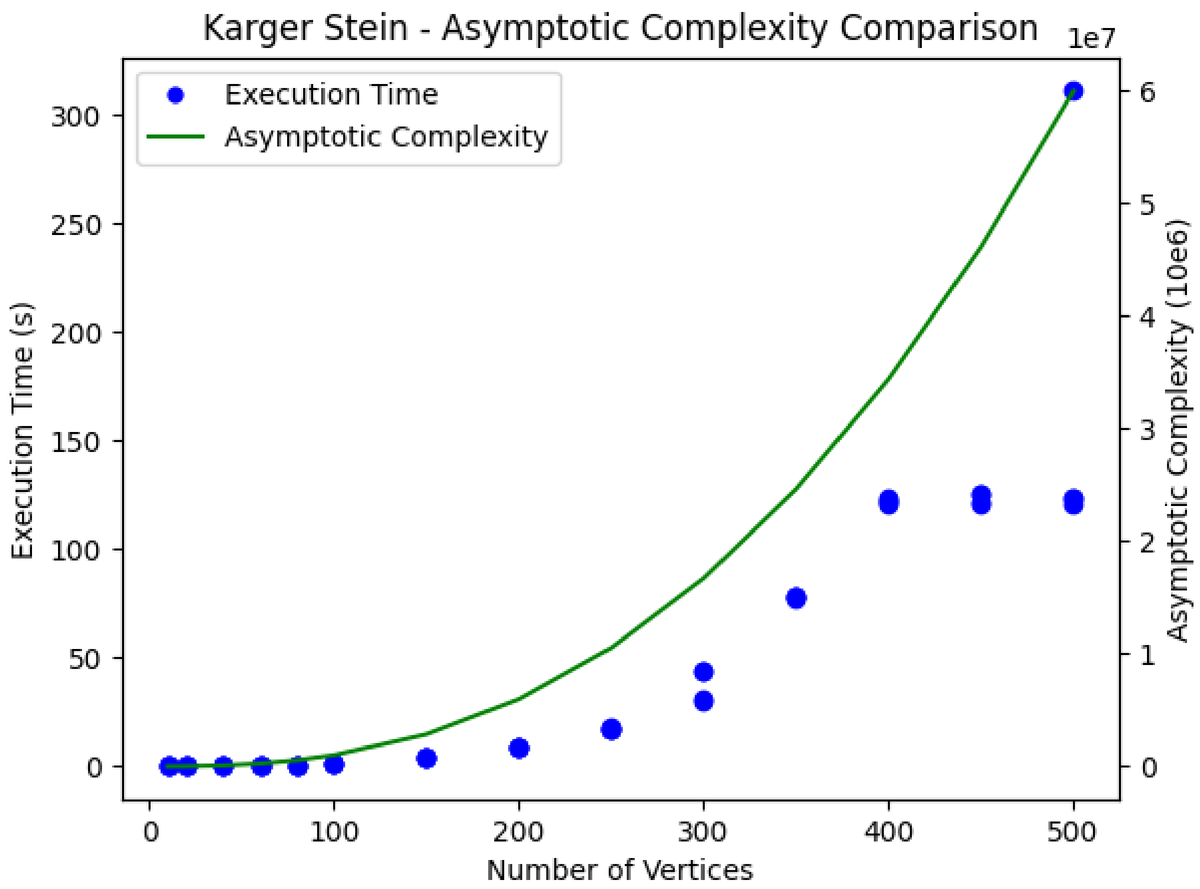

Figure 6: Karger and Stein's Time Cost compared to the Asymptotic Complexity

5 Conclusion

As we can see from the results table, the execution time of Karger Stein greatly increases at 100 nodes, going from less than one second to complete the algorithm to over four seconds. The execution time exponentially grows, and eventually starts to time out at the implemented time limit of 120 seconds once the graph exceeds 350 nodes. The execution time of Stoer Wagner also increases as the nodes increase, but not as dramatically as Karger Stein. Even with the largest file of 500 nodes, the Stoer Wagner algorithm takes less than a second to complete. Due to the large difference in the execution time, and considering that Stoer Wagner is a deterministic algorithm, we can conclude that Stoer Wagner is not only better than Karger Stein but is also more efficient.