The Minimum Cut Problem For An Undirected Graph

ABLORDEPPEY Prosper

***Abstract* –In this context, given an undirected graph $\mathcal{G}(\mathcal{V},\mathcal{E})$, with $n = |\mathcal{V}|$ vertices and $m = |\mathcal{E}|$ edges, a minimum cut of $\mathcal{G}$ is a partition of the graph's vertices into two complementary sets S and T, such that the number of edges between the set S and the set T is as small as possible. This presentation studies this problem by designing and testing an exhaustive search algorithm to solve it, as well as an algorithm using a greedy heuristics. A probabilistic randomized technique was also proposed, analyzed and improved so as to obtain the exact minimum cut of interest.**

I. Notation and Problem Definition

Consider an undirected graph $\mathcal{G}(\mathcal{V},\mathcal{E},\mathcal{W})$ with $n = |\mathcal{V}|$ vertices and $m = |\mathcal{E}|$ edges where $w(\{u,v\}) \in \mathcal{W}$ is the number of edges connecting vertex u and vertex v for $(u,v) \in \mathcal{E}$.

- Let $Adj(\mathcal{G})$ be the adjacency matrix of graph $\mathcal{G}$. $Adj(\mathcal{G})$ is a two dimensional (2D) matrix of ones and zeros where a one (1) indicates the presence of the corresponding edge in the network [2]. Suppose a_{ij} is an entry of the $Adj(\mathcal{G})$. Then

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from vertex } j \text{ to vertex } i \\ 0 & \text{otherwise} \end{cases}$$

- Let $vset = \{S,T\}$ be a complementary partition of $\mathcal{V}$ then
$$S \cup T = \mathcal{V}, S \cap T = \emptyset$$
- We can define the set of cut edges as
$$C = \{\{u,v\} \in \mathcal{E} | u \in S, v \in T\}$$
- The cut size
$$\Theta(vset) = \sum_{\{u,v\}\in C} w(\{u,v\})$$
which is equal to the cardinality of C if all edges have unitary weights but different otherwise.
- A group of vertices collapsed into one (1) is termed as a *supernode*.
- An edge connecting from a supernode to another vertex is called, a *superedge*.
- The weight of a superedge is the sum of weights of all individual vertices connecting both supernodes.
- All graphs $\mathcal{G}$ considered in this presentation are connected. Thus, there exists at least one (1) edge between any vertex of $\mathcal{V}$.
- Degree of a vertex u represented as $Deg(u)$ is the number of edges connecting vertex u to its immediate neighbors.

II. Outline Of Implementation

The outline of this project is given as follows

(III) describes the Brute-Force or Exhaustive-Search implementation.
(IV) describes the implementation of a Greedy-Heuristic technique.
(V) implements a random probabilistic approach to estimate the minimum cut. An improved implementation which offers a higher chance of realizing the exact minimum cut compared to the naive method was also proposed.
(VI) complexities analysis.
(VII) auxiliary functions

III. Brute-Force / Exhaustive Implementation

This technique computes the *cut* for all possible complementary partitions of the vertices set $\mathcal{V}$. The minimum of all these computed cuts represents the *minimum cut* of the graph $\mathcal{G}$. The implementation is presented in Algorithm (1) below with two helping methods **GetPartCut** and **GetCompPartitions** described below in detail.

A. Compute Partition Cut

Algorithm 1: Compute Cut Of A Partition of $\mathcal{V}$

```
Input: $Adj(\mathcal{G})$ ;            // adjacency matrix of $\mathcal{G}$
    1 $S$ ;        // $1^{st}$ partition element of $vset$
    2 $T$ ;        // $2^{nd}$ partition element of $vset$
Output: Cut
3 Function GetPartCut:
4     $Num\_Cuts \leftarrow 0$
5     foreach $v_1 \in$ S do
6         foreach $v_2 \in$ T do
              // check if $(v_1, v_2) \in \mathcal{E}$
7             if $edge([v_1, v_2]) \in Adj(\mathcal{G})$ then
8                 $Num\_Cuts = Num\_Cuts + 1$
9     return $Num\_Cuts$
```

This function returns the cut for a given complementary partition. There are a few other ways to generate the cut of a complementary partition. The logic behind most of them is to contract/merge all vertices of each partition set into supernodes. The weight/cut of the superedge connecting both supernodes then represents the sum of the weights of all contributing edges to some node.

The one proposed in this project made use of the adjacency matrix $(Adj(\mathcal{G}))$, which happens to be very straight forward and has easy implementation. The cut for a partition $vset$ is the number of edges connecting S and T. To achieve this, we leverage the fact that if an edge exists between two vertices (v_1, v_2), then, their representation/value in the adjacency matrix is one (1). If this is achieved, the number of cuts (Num_Cuts) is updated by one (1).

The cut for the given complementary partition will be returned after the end of execution of Algorithm (1). This algorithm runs in $\mathcal{O}(|S| \cdot |T|)$ time. The complexity for this algorithm is analyzed below for the two possible cases where the number of vertices n is even or odd.

- If $|\mathcal{V}| = n$ is even

$$\begin{aligned} C =& \mathcal{O}\left(|S| \cdot |T|\right) \\ =& \mathcal{O}\left(\frac{n}{2} \cdot \frac{n}{2}\right) \\ =& \mathcal{O}\left(\frac{n^2}{4}\right) = \mathcal{O}(n^2) \end{aligned}$$

- If $|\mathcal{V}| = n$ is odd

$$\begin{aligned} C =& \mathcal{O}\left(|S| \cdot |T|\right) \\ =& \mathcal{O}\left(\frac{n-1}{2} \cdot \frac{n}{2}\right) \\ =& \mathcal{O}\left(\frac{n(n-1)}{4}\right) \\ =& \mathcal{O}\left(\frac{n^2-n}{4}\right) = \mathcal{O}(n^2) \end{aligned}$$

Hence, for all cases of n the complexity of this algorithm is $\mathcal{O}(\mathbf{n^2})$

B. Generate Valid Complementary Partitions

Algorithm 2: Get Valid Compl. Partitions Of $\mathcal{V}$

```
Input: $\mathcal{V}$                                  // vertices set
Output: CompParts    // complementary partitions
1  Function GetCompPartitions:
2      Set $CompParts \leftarrow []$
3      Set $\mathcal{P} \leftarrow$ Powerset($\mathcal{V}$)            // length $2^{|\mathcal{V}|}$
4      foreach $subset \in \mathcal{P}$ do
           // ensures no empty partition
5          if $(subset \neq \emptyset)$ or $(subset \neq \mathcal{V})$ then
6              $S = subset$
7              $T = \mathcal{V} \setminus S$
8              CompParts.insert($(S, T)$)
       // remove permutations of partitions
9      CompParts = CompParts.$remove\_perms()$
10     return CompParts
```

The goal of this function is to generate all non-empty, non-permuted complementary partitions of the given vertices set $\mathcal{V}$. We can easily obtain such partitions by considering the powerset ($\mathcal{P}$) of $\mathcal{V}$. The function returns a list of tuples, each comprising of a subset and its complement in V. The total of all these subsets are 2^n. Since our interest is to obtain non-empty partitions, we eliminate the empty set ($\emptyset$) and the set with the full vertices (have complement as the empty set $\emptyset$). This results in $(2^n - 2)$ partitions returned by this function. The complexity of this algorithm runs in $\mathcal{O}(\mathbf{2^n})$.

Again, a basic analysis of all complementary sets returned by this function contains symmetry partitions. To illustrate, lets consider a connected graph with three (3) vertices. Thus, $\mathcal{V} = \{0, 1, 2\}$. From the derived expression above, the number of complementary sets will be $|CompParts| = 2^3 - 2 = 8 - 2 = 6$.

TABLE I
VSETS FOR $\mathcal{V}$

#	S	T
1	{0}	{1,2}
2	{1}	{0,2}
3	{2}	{0,1}
4	**{0,1}**	**{2}**
5	**{0,2}**	**{1}**
6	**{1,2}**	**{0}**

As we can clearly observe from TABLE I, half of these complementary partitions are simply permutations of the second half. Considering larger vertices set, this phenomenon also applies. Hence its prudent to restrict only half of these initial number of complementary sets. Thus, the actual number of non-permuted complementary sets generated for the above example will be

$$|CompParts| = \frac{2^3 - 2}{2} = \frac{8-2}{2} = 3$$

The general number of complementary partitions that will be generated given $n = |\mathcal{V}|$ is then given by

$$|CompParts| = \frac{2^n - 2}{2} = (2^{n-1} - 1)$$

This goes a long way to decrease the running time of the second part of the brute-force or exhaustive search algorithm implemented. Since cuts will be computed only for $(2^{n-1} - 1)$ different partitions instead of the previous $(2^n - 2)$ we saw earlier.

C. Main Exhaustive Search Method

First, we obtain all the various complementary partitions of the vertices set using the **GetCompPartitions** method as described above. To get the min_cut_size of the graph, we could consider two (2) approaches;

(1) Create an array to hold the cuts for all partitions. We compute the cut for each partition and append it to the array. After that, we find the minimum value of the array. This represents our min_cut_size.

Algorithm 3: Exhaustive Search

```
1  [h]  Input: $\mathcal{G}(\mathcal{V},\mathcal{E})$      // Graph nodes $\mathcal{V}$ edges $\mathcal{E}$
   Output: $min\_cut\_set$       // minimum cut set of $\mathcal{G}$
 2 $min\_cut\_size$              // minimum cut size of $\mathcal{G}$
3  Function ExhaustiveSearch:
4
5      Set $min\_cut\_size \leftarrow \infty$
6      Set $min\_cut\_set \leftarrow None$
7      Set $CompParts \leftarrow$ GetCompPartitions($\mathcal{V}$)
8
9      foreach $S, T \in CompParts$ do
10         $cut\_size \leftarrow$ GetPartCut($Adj(\mathcal{G}), S, T$) if
             $cut\_size < min\_cut\_size$ then
11               $min\_cut\_size = cut\_size$
12               $min\_cut\_set = (S, T)$
13     return $min\_cut\_set, min\_cut\_size$
```

(2) instantiate a very large integer value to be the min_cut_size. For every cut computed for each complementary partition, we compare this cut to the min_cut_size value. If the cut is smaller, then we update the min_cut_size with the computed cut, otherwise, we continue to compute the cut for the next partition.

Analysis of the first approach is computationally expensive due to IO overhead. The overhead results from array creation; writing cuts to array and the searching for the minimum value from the array. This leaves us with the second approach which is much more efficient. The second technique was implemented as presented in the Algorithm (1).

As highlighted above, the complexity of the **GetCompPartitions** method is $\mathcal{O}(2^n)$, which returns an array of $(2^{n-1} - 1)$ complementary partitions. For each of these complementary partitions, the **GetPartCut** algorithm is called which also runs in $\mathcal{O}(\mathbf{n^2})$. Putting everything together, the Exhaustive Search implementation of the minimum cut problem runs in complexity

$$\begin{aligned} C &= \mathcal{O}\left(2^n + (2^{n-1} - 1) \cdot n^2\right) \\ &= \mathcal{O}\left(2^n + n^2 \cdot 2^{n-1} - n^2\right) \\ &= \mathcal{O}\left(2^n + n^2 \cdot 2^{n-1}\right) \\ &= \mathcal{O}\left(\mathbf{2^n}\right) \end{aligned}$$

Thus Exhaustive Search will probably not be used until faster methods have first reduced the size of the problem.

IV. Greedy-Heuristics Implementation

A Greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage. [1] In many problems, a greedy strategy does not produce an optimal solution, but a greedy heuristic can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

In this specific minimum cut problem, a proposed implementation of the greedy heuristics is to pick the vertex v with the smallest number of edges connecting to it and return the number of edges connecting to v as the minimum cut for our graph. More formally, we compute the cut as the vertex with the minimum degree in the graph. The cut computed is returned hoping it is the true minimum cut of the graph, which is the basic logic behind the greedy search algorithm. Another construction of this technique is to compute the degree of all the vertices of the graph. The smallest degree vertex $min(\{Deg(v); \forall v \in \mathcal{V}\})$ in the graph is determined. This vertex forms one of the partition elements S. All the other vertices forms the second set of the complementary partition T. This algorithm estimates the actual minimum cut for some instances, but for some carefully chosen problems, this algorithm tends to report inaccurate minimum cut.

Fig (1 and 2) below shows an illustration of the greedy-heuristics for the minimum cut problem in action. In Fig 1, luckily for us, the algorithm returned the correct minimum cut to the problem. But when instead we had a carefully designed graph as presented in Fig 2, our implementation resulted in an incorrect solution where the minimum cut size reported is three (3) corresponding to the complementary set {(0),(1,2,3,4,5,6,7)}. The correct minimum cut size which the Exhaustive-Search will report will be two (2) corresponding to the partition {(0,1,2,3),(4,5,6,7)}.

It is also worth noting that the worst cut size to be reported by this technique will be $n = |\mathcal{V}|$ in the case of a complete graph which happens to be the actual minimum cut in such scenarios.

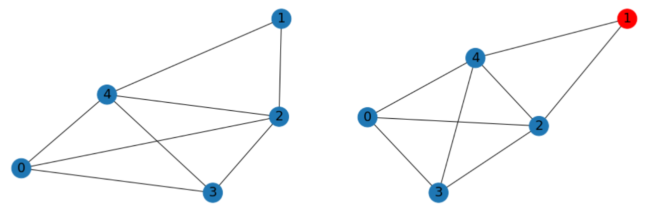


Fig. 1 - Correct Greedy Estimate

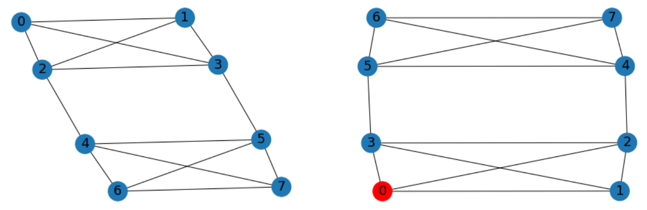


Fig. 2 - Wrong Greedy Estimate

The greedy algorithm will run more quickly than the exhaustive one. But the greedy algorithm does not guarantee an optimal solution to the problem. As analyzed above.

A. Compute Vertex Degree $Deg(u)$

This method computes the degree of a given vertex via the adjacency matrix. In this case, for a given vertex, we search

Algorithm 4: Compute Degree Of Vertex

```
  Input: $Adj(\mathcal{G})$            // adjacency matrix of $\mathcal{G}$
1 $vertex$                              // $vertex \in \mathcal{V}$
  Output: $degree$                      // degree of vertex $\in \mathcal{G}$
2 Function GetVertexDegree:
3     Set $degree \leftarrow 0$
      // fetch row of vertex
4     foreach $v_2 \in Adj(\mathcal{G})[vertex]$ do
          // if edge exists
5         if $v_2 == 1$ then
6             $degree = degree + 1$
7     return $degree$
```

through the remaining $(n-1)$ vertices which are on the columns of the adjacency matrix and where we have a one (1) value, then we have a neighbor of the vertex. So we count all the immediate neighbors of the given vertex and return the result. Clearly, since we do not have self loops for vertices in any arbitrary graph for this project, we search for neighboring vertices among the remaining $(n-1)$ vertices. Thus, our above algorithm runs in complexity $\mathcal{O}(\mathbf{n-1}) = \mathcal{O}(\mathbf{n})$.

B. Main Greedy Heuristic Method

Algorithm 5: Greedy Heuristics

```
  Input: $\mathcal{G}(\mathcal{V},\mathcal{E})$     // Graph vertices $V$ edges $E$
  Output: $min\_cut$                    // minimum cut of $\mathcal{G}$
1 Function GreedyHeuristics:
2     Set $min\_cut\_size \leftarrow \infty$
3     Set $min\_cut\_set \leftarrow None$
4     foreach $vertex \in \mathcal{V}$ do
5         $cut\_size \leftarrow$ GetVertexDegree($vertex, \mathcal{A}(G)$)
6         if $cut\_size < min\_cut$ then
7             $min\_cut\_size = cut\_size$
8             $min\_cut\_set = (vertex, \mathcal{V} \backslash vertex)$,
9     return $min\_cut\_set, min\_cut\_size$
```

This GreedyHeuristics method implements a search for the minimum cut which may be the actual value or not. For each vertex in the vertices set, a cut is computed as the number of immediate neighbors of the vertex (degree of vertex) in question. If the computed degree is smaller than the min_cut_size value, then min_cut_size value is replaced with the computed degree. The min_cut_size value which represents the vertex with the minimum degree is returned hoping it is the actual minimum cut.

There are n vertices in the vertices set $\mathcal{V}$ and for each vertex, the **GetVertexDegree** method is called to compute the degree for the vertex in question which runs in $\mathcal{O}(\mathbf{n})$, the complexity for this GreedyHeuristic technique implemented becomes

$$\begin{aligned} C =& \mathcal{O}(n \cdot n) \\ C =& \mathcal{O}(\mathbf{n^2}) \end{aligned}$$

V. Probabilistic And Randomized Implementation

In this module, unlike the ExhaustiveSearch technique which returns the exact minimum cut by considering all possible cuts of the graph, a random complementary partition is selected in a uniform fashion out of the $(2^n - 2)$ different partitions. Since we know half of these partitions set are permutations, then the probability of realizing the exact minimum cut is at least

$$\begin{aligned} \mathcal{P} =& \left(\frac{2}{2^n - 2}\right) \\ =& \frac{1}{2^{n-1} - 1} \end{aligned}$$

Reason simply being that, there exists a permutation of the actual minimum cut complementary partition.

The implementation of this algorithm takes two (2) major steps. The idea is to randomly choose an edge from the graph and merge the vertices into a supernode, whilst updating the weight of the superedges connecting the newly formed supernode and neighbours of contributing vertices. This is repeated until a superedge remains in the graph connecting two (2) supernodes. The weight of this superedge represents the minimum cut of the graph.

This algorithm utilizes two (2) functions

- Picking random edges to merge. $Algorithm$ (6)
- Updating the Graph. $Algorithm$ (7)

A. Merging Operation

From Algorithm 6, we realize that the edge contracting/merging is done on one (1) edge at a time, until only one edge remains in the graph. In the worst case, the neighbors of a vertex set could be $(n-1)$ which runs in $\mathcal{O}(n-1) = \mathcal{O}(\mathbf{n})$.

B. Main Probability Randomized Method

The **ProbabilityRandomized** in Algorithm 7 is the actual implementation of a randomized algorithm for the minimum cut. The algorithm merges vertices of a random edge one (1) at a time until only two (2) vertices are remaining. This runs in complexity $\mathcal{O}(n-2) = \mathcal{O}(n)$. The **UpdateGraph** method in Algorithm 6 runs at most for $(n-1)$ vertices. Thus, $\mathcal{O}(n)$ time. Putting all together, this algorithm runs in $\mathcal{O}(n \cdot n) = \mathcal{O}(\mathbf{n^2})$

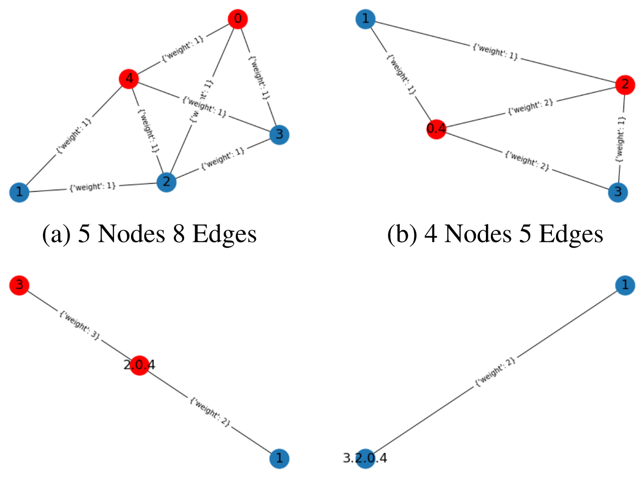
On multiple runs of this algorithm, the minimum cut returned may be different from the previous estimate.

Fig (3) is an implementation of the randomized algorithm as described in **ProbabilityRandomized** in Algorithm 7 above on a connected graph with five (5) vertices and eight (8) edges. The default weight of all edges in the graph is one (1). In each instance of the execution, an edge is picked at random and merged into a supernode, until only one edge is left. From the start to the finish of the algorithm, it is worth noting that, $(5-2) = 3$ vertices were

```
Algorithm 6: Merge Edge Update Graph
Input: $\mathcal{G}(\mathcal{V},\mathcal{E})$                 // graph to update
  $(u, v)$                                  // edge to merge $\in \mathcal{E}$
Output: $\mathcal{G}(\mathcal{V},\mathcal{E})$          // updated graph $\mathcal{G}(\mathcal{V},\mathcal{E})$
Function UpdateGraph:
    Set $supernode \leftarrow$ string('$u \cdot v$')
    Set $updated\_graph \leftarrow$ empty_graph

    // fetch the neighbors of $u$ and $v$
    u_neighbors = $\mathcal{E} \cdot neighbors(u)$
    v_neighbors = $\mathcal{E} \cdot neighbors(v)$

    // create superedge for $u$ edges
    foreach $nb \in u\_neighbors$ do
        if $nb <> v$ then
            edge_weight = $w(\{u, nb\} \in \mathcal{E})$
            $\mathcal{E}\cdot$ add$(nb, supernode, edge\_weight)$

    // create superedge for $v$ edges
    foreach $nb \in v\_neighbors$ do
        if $nb <> u$ then
            // edge_weight wt
            wt = $w(\{v, neighbor\} \in \mathcal{E})$
            // update superedge's weight
            if $(nb, supernode) \in \mathcal{E}$ then
                $\mathcal{E}(nb, supernode).wt+ = wt$
            else
                $\mathcal{E}\cdot$add$(nb, supernode, edge\_weight)$

    // edges updated, remove initial nodes
    $\mathcal{E} \cdot remove\_edge(u, v)$
    return $\mathcal{G}(\mathcal{V},\mathcal{E})$
```

```
Algorithm 7: Randomized Algorithm
Input: $\mathcal{G}(\mathcal{V},\mathcal{E})$       // Graph vertices $\mathcal{V}$ edges $\mathcal{E}$
Output: $min\_cut\_size$  // minimum cut size of $\mathcal{G}$
  $min\_cut\_set$                 // minimum cut edges
Function ProbabilisticRandomized:
    while $|\mathcal{E}| > 1$ do
        // choose random edge to merge
        edge_to_merge = uniform·random$(\{e; e \in \mathcal{E}\})$
        $\mathcal{G}(\mathcal{V},\mathcal{E},\mathcal{W})$ =
          UpdateGraph$(edge\_to\_merge, \mathcal{G}(\mathcal{V},\mathcal{E},\mathcal{W}))$
    $min\_cut\_set = (u, v) \in \mathcal{E}$
    $min\_cut\_size = \{w(u, v) \in \mathcal{W}\}$
    return $min\_cut\_set, min\_cut\_size$
```

merged on. Also, the example happens to give the correct minimum cut, but only because we carefully picked the edges for merging. There are many other choices of edges for merging, so it's possible we could have ended with a cut with more than two (2) edges.



(a) 5 Nodes 8 Edges (b) 4 Nodes 5 Edges

(c) 3 Nodes 3 Edges (d) 2 Nodes 1 Edge

Fig. 3 - Random Edge Contraction

C. *Improving Randomized Results*

From the literature [3], running the **ProbabilityRandomized** algorithm multiple times and returning the minimum cut results in a much higher probability of realizing the exact minimum cut. It was proved that, the number of trials to perform can be expressed as $\left(\mathcal{C} \cdot \binom{n}{2} \cdot \ln(n)\right)$ so as to have a higher chance of realizing the minimum cut with probability of at least

$$\mathcal{P} \geq 1 - \frac{1}{n^C}$$

The improved method is presented in Algorithm 8, with complexity,

$$\begin{aligned} C &= \mathcal{O}\left(\mathcal{C} \cdot \binom{n}{2} \cdot \ln(n) \cdot n^2\right) \\ &= \mathcal{O}\left(\frac{n(n-1)}{2} \cdot n^2 \cdot \ln(n)\right) \\ &= \mathcal{O}\left(n^4 \cdot \ln(n)\right) \end{aligned}$$

```
Algorithm 8: Improved Randomized Algorithm
Input: $\mathcal{G}(\mathcal{V},\mathcal{E})$      // Graph vertices $\mathcal{V}$ edges $\mathcal{E}$
Output: $min\_cut\_size$  // minimum cut size of $\mathcal{G}$
  $min\_cut\_set$                 // minimum cut edges
Function ImpProbabilisticRandomized:
    $min\_cut\_size = \infty$
    $min\_cut\_set = None$
    Num_Trials $= \mathcal{C} \cdot \binom{n}{2} \cdot \ln(n)$     // $\mathcal{C} \in \mathcal{N}, n = |\mathcal{V}|$
    foreach $trial \in Num\_Trials$ do
        cut_set, cut_size =
          ProbabilisticRandomized$(\mathcal{G}(\mathcal{V},\mathcal{E},\mathcal{W}))$
        if $min\_cut\_size < cut\_size$ then
            $min\_cut\_size$ = cut_size
            $min\_cut\_set$ = cut_set
    return $min\_cut\_set, min\_cut\_size$
```

For simplicity sake, I chose the value of $\mathcal{C} = 1$ which could be any positive integer.

VI. Computational Complexities

For analysis purposes, only some graphs were selected for comparison of the various algorithms.

A. *Graphs Considered For Analysis*

The graphs considered are presented in TABLE II. From observation, any graphs with vertices larger than twenty (20) takes infinitely long time to execute the Exhaustive-Search algorithm. The choice of the various graphs is done so as to realize some patterns among the executions of the various algorithms.

TABLE II
Graphs Considered

Graph	Vertices	Edges
1	5	8
2	10	28
3	15	64
4	20	108
5	25	168

B. *Algorithms Complexities*

It can be observed from Fig. 4 that, the complexity of the Exhaustive-Search algorithm for graphs having more than eighteen (18) vertices tend to increase very fast, as compared to the other algorithms. This explains why my computer (*16GBRAM, AMD Ryzen 7 4800H with Radeon Graphics and speed 2.90 GHz*) had to restart automatically due to the expensive computational resource used up by the Exhaustive-Search algorithm when computing the minimum cut for graph with twenty-five (25) vertices. The Improved Probability Randomized algorithm has a reasonable running time compared to the Exhaustive Search. Since the Greedy and Probabilistic Randomized algorithms run have the same complexities, we didn't observe any significant difference in their running times.

After being able to compute the minimum cut for graphs with at most twenty (20) vertices, which involves about one-million (1,500,000) iterations of the Exhaustive-Search method, any higher vertice graphs considered have undoubtedly high number of iterations. For instance, over a billion iteration is expected for a graph with thirty (30) vertices, which could greatly impact the performance of the running system and may cause a break down.

C. *Algorithms Timed Executions*

TABLE IV reports a timed implementation of the algorithms which confirm the analysis of complexities conducted in the previous section; with the Greedy and Probabilistic Random algorithm performing similarly. As discussed during the complexities analysis, graph five (5) never completed with my computer breaking down on the Exhaustive-Search implementation. The improved Improved Probabilistic Random technique was able to obtain the correct minimum cut in less times as compared to the Brute-Force technique.

TABLE III
Complexities

Brute-Force	Greedy	ProbRand	ImpProbRand
$\mathcal{O}\left(2^n\right)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}\left(n^4 \cdot \ln(n)\right)$

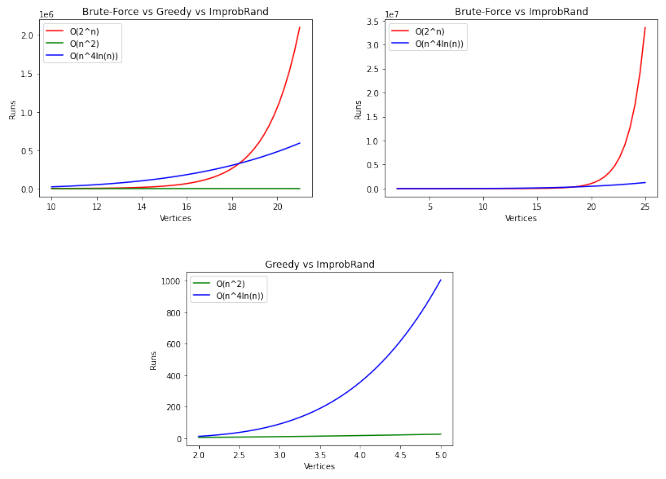


Fig. 4 - Graph Of Complexities

TABLE IV
Timed Executions (sec)

Graph	Brute-Force	Greedy	ProbRand	ImpProbRand
1	0.003	0.0	0.0	0.002
2	0.082	0.0	0.0	0.034
3	3.934	0.0	0.001	0.207
4	171.034	0.001	0.001	0.739

VII. Auxiliary Functions

We employed four (4) functions to help with the implementation of this project. They are;

1) CreateDirectory
   - create directories to hold graphs and plots.
   - deletes directories as requested.
2) GenerateGraphs
   - generates random Erdos Renyi graphs based on specified parameters.
   - plot graphs after generation

- save created graph to directory in current graphs directory.

3) LoadGraph
    - load a graph from file.
    - plot the graph if requested.
4) PlotGraph
    - main implementation of graph plotting.
    - weighted or not weighted.
    - partly colored or not.

References

[1] Paul E Black. greedy algorithm, dictionary of algorithms and data structures. *US Nat. Inst. Std. & Tech Report*, 88:95, 2012.

[2] Nykamp DQ. Adjacency matrix definition.

[3] Anthony Kim. Min cut and karger's algorithm : Min cut and karger's algorithm, 2016.