An Approximation Algorithm for Vertex Cover Problem

Nishant Jain
Student, School of Computer Science
D.A. University, Indore, M.P, India

Shipra Shukla
Student, School of Computer Science
D.A. University, Indore, M.P, India

Abstract- The Vertex Cover problem fascinates computer scientists and surfaces in various real world applications. This problem has been proved NP complete in recent future. Therefore one needs to look for polynomial time approximation algorithms to solve the problem. Many algorithms have been developed yet which can find an approximate answer to the problem. We have designed an approximation algorithm to solve vertex cover problem. This algorithm is tested on a large number of graphs adopted from literature [1]. Proposed approximation algorithm yields better covers than 2-approximation algorithm. Proposed algorithm offers superior running time than brute force strategy.

Keywords- Minimum vertex cover, Approximation algorithm, Breaking graph, Rejoining etc.

I. INTRODUCTION

Richard Karp presented a list of 21 NP-complete problems in his paper "Reducibility among Combinatorial Problems" in 1972. One of these 21 NP-complete problems to appear in the list is the vertex cover problem. In computational complexity theory it is a classical NP-complete problem; therefore it is unlikely to be solved in polynomial time for the worst case problem instance. The minimum vertex cover problem includes graph theory and finite combinatory.

In a given graph, we have to find a smallest set of vertices such that every edge of the graph has at least one end vertex in the set. It covers all the edges of the given graph. Even though the problem is NP-complete, it can be solved in polynomial time for bipartite graphs and tree graphs, but for worst case graphs polynomial time algorithms are unlikely to exist.

Two versions of vertex cover problem are the decision version and the optimization version. In the decision version, the goal is to verify for a given graph whether there exists a vertex cover of a specified size or not. On the other hand, in the optimization version of this problem, the goal is to find a minimum vertex cover out of all feasible covers. Our algorithm is working on the second one i.e. optimization problem.

II. PROPOSED ALGORITHM (INFORMAL DESCRIPTION)

The algorithm works in two phases:

1. Breaking the graph-

In the breaking phase we select a set of vertices on the basis of their degree. Choose a maximum degree vertex then remove this vertex and corresponding edges from the graph. Store this vertex into a stack and also maintain a record of its neighbors. Now choose a neighbor of this maximum degree vertex and delete this neighbor vertex and corresponding edges. Store it into a stack and maintain a record of its neighbors. Again choose the maximum degree vertex and repeat the process until the graph becomes edgeless. Now we have a set of vertices stored in a stack. This set will be used in the rejoining phase.

2. Rejoining-

Selected set of vertices are again applied to the graph in reverse order to find the vertex cover. Initially the cover is empty. Pick a cover from the list of cover nodes and delete it from the list. Add a vertex from the top of the stack to the and check the following three conditions:

i) If all the neighbors (n) of the vertex are present in the cover then do not add it to the cover.

ii) If (n-1) neighbors are present in the cover then either add the node or neighbor to the cover. Now it will generate two covers – one having the node and other having the neighbor. Store the second cover into a list of cover nodes.

iii) If less than (n-1) nodes are present in the cover then add the vertex to the cover.

Repeat this process until the whole set of vertices becomes empty and show the generated cover as output. Now pick the second cover and delete it from the list of cover nodes. Apply all the vertices from the selected set on this cover and compare the generated answer to the previous one. If the second cover is less than the previous cover, show it as output. Repeat the process for limited number of times according to our need. If we increase the limit then more number of cover nodes will be processed and we will get better answer but processing time will increase.

This algorithm will generate number of outputs and each time the generated output will be better than the previous output.

Code

```
int VertexCover(const Graph& a)
{
  Vertex* b = new Vertex[N];
   Graph temp = a;

   int j = 0; int odd = 1;
   while(temp.NodeCount()!=0){
      if(odd==1){

      int id = temp.MaxDegreeVertex();

      b[j]= temp.GiveVetexInfo(id);

      j++;
      temp.RemoveVertex(id);
      odd = 0;
   }
   else{
      int id = b[j-1].Adjacent[0];
      b[j]= temp.GiveVetexInfo(id);
      j++;
      temp.RemoveVertex(id);
      odd = 1;
   }
   }
CoverNode p;
p.Size = 0;
p.j = j;
Stack C;
C.Push(p);
CoverNode q;
int Min = N;
int counter=0;
while(C.StackEmpty()!=1) {
          if(counter==10)          // Increase the value of counter to get better result
```

```
            break;
            counter++;
             q = C.Pop();
  (q.j)--;
  int j = q.j;
   while(j >=0 ){
          if(q.Size>=Min)
          break;
          int ee = Min - q.Size;
          if((j/2)>ee)
          break;
          Apply(q,j,b,C);
          j--;
         }
   if(q.Size < Min){
      if(j==-1){
      p = q;
      Print(&p);
      Min = q.Size;
      }
   }
}
Print(&p);
  }
```

III. ANALYSIS

The analysis performed on the proposed algorithm shows that it is a polynomial time algorithm. As we take a look on the breaking phase, the number of time it will break the graph can't be greater than the total number of nodes in the graph. Because the loop is running till the node count is zero. So the running time of the processes will be polynomial.

In the rejoining phase we will add the previously selected set of vertices to the vertex cover. There are two loops working for this phase. Number of iterations for the first loop will be 10 because we are taking first 10 covers from the vertex cover list for processing. For second loop the number of iteration will be equal to the number of vertices in the selected set. The number of selected vertices will always be less than the total number of vertices. So this phase also generates a polynomial time solution. Thus we can conclude that the given algorithm is a polynomial time algorithm.

The proposed algorithm is much better than 2- approximation algorithm. In the breaking phase we are selecting the vertices on the basis of 2- approximation with some modifications (taking the max degree vertex and its neighbor each time). So it will generate a reduced set of vertices than that of 2- approximation. Again we are reducing this set in the rejoining phase to find a better vertex cover. Thus, the produced cover will be much better than 2-approximation.

We have discussed here some key points that we have used for the development of algorithm. We have selected maximum degree vertex for breaking the graph so that the graph can be broken easily. We have applied top down approach for breaking the graph and bottom up approach for reconstruction. We have applied the concept of partial divide and conquer technique.

IV. CONCLUSION

The proposed algorithm is giving the optimal answer for most of the graphs. For small graphs the algorithm is giving almost exact answer and for large graph the answer is not exact but very close to it. The algorithm will give better answer if we increase the limit of counter variable. As we increase the limit of counter variable it will process more number of vertex covers and find a better answer than the previous one.

V. ACKNOWLEDGEMENT

We have no words to express our gratitude towards my guide Dr. Deepak Abhyankar, Software Engineer, School of Computer Science And Information Technology, DA University, Indore who has provided all the essential material for the research paper. His guidance and motivation encouraged us and provided us an idea on how to work for this paper. We are highly thankful to him.

REFERENCES

[1] http://www.dharwadker.org/vertex_cover/,2016
[2] https://en.wikipedia.org/wiki/Vertex_cover,2016
[3] http://www.dharwadker.org/pirzada/applications/2016
[4] R.M. Karp, Reducibility among combinatorial problems, Complexity of Computer Computations, Plenum Press, 1972.
[5] Stanley Lippman, Essential C++, Addison-Wesley, 2000.
[6] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. (1990). Introduction to Algorithms (first ed.). MIT Press and McGraw-Hill. ISBN 978-0-262-03141-7.

APPENDIX

```
#include <iostream>
#include<fstream>
#include<string>
using namespace std;
 int N;
 int stop;
struct Vertex{
    int vertexId;
    int neighbourCount;
    int* Adjacent;
   Vertex& operator= (const Vertex& other){
      if(this!=&other){
         vertexId = other.vertexId;
         neighbourCount = other.neighbourCount;
         Adjacent = new int[neighbourCount];
         int j = 0;
         while(j<neighbourCount){
            Adjacent[j]=other.Adjacent[j];
            j++;
        }
     }
     return *this;

  }
};


class Graph{
    int vertexCount;
Vertex* a;
   public:
      Graph(ifstream& File){
         int number;
      vertexCount = N;
      a = new Vertex[N];
      int j = 0;
      int count;
      while(j <N){
     count = 0;
      a[j].vertexId = j;
       a[j].Adjacent = new int[N-1];
       int k = 0;
         while(k < N){
         File >> number;
         if(number==1){
            a[j].Adjacent[count] = k;
            count++;
         }
         k++;

         }
       a[j].neighbourCount = count;

         j++;

      }


}
```

```
 int NebCount(int id) const{
    return a[id].neighbourCount;
  }

  int NodeCount() const{
    return vertexCount;
  }

  Graph(const Graph& ob){
    vertexCount = ob.vertexCount;
     a = new Vertex[N];
    int j = 0;
    while(j<N){
       a[j].neighbourCount =
ob.a[j].neighbourCount;
       a[j].vertexId = j;
       int count = a[j].neighbourCount;
       a[j].Adjacent = new int[count];
       int k = 0;
       while(k < count){
          a[j].Adjacent[k] = ob.a[j].Adjacent[k];
          k++;
       }
       j++;
    }
  }

  void RemoveEdge(int c,int  d){
  int count = a[c].neighbourCount;
  int j = 0;
  while(j<count){
    if((a[c].Adjacent[j])==d)
    break;
    j++;
  }
  a[c].Adjacent[j] = a[c].Adjacent[count-1];
  a[c].neighbourCount--;
  if(a[c].neighbourCount==0){
    delete a[c].Adjacent;
  vertexCount--;
  }
   }

 void RemoveVertex(int id){
  int count = a[id].neighbourCount;
  int j = 0;
  while(j < count){
    int neb = a[id].Adjacent[j];
    RemoveEdge(neb,id);
      j++;
  }
  if(count >0){
  a[id].neighbourCount = 0;
  delete a[id].Adjacent;
  vertexCount--;
  }

 }
int GiveNeighbour(int id) const{
        return a[id].Adjacent [0];

 }

  int LeastDegreeVertex() const{
   int count = N;
   int j = 0;
   int result = -1;
   while(j < N){
     if(a[j].neighbourCount){
       if((a[j].neighbourCount)<count){
       result=j;
       count = a[j].neighbourCount;
     }
     }
     j++;
   }
   return result;
  }
   int MaxDegreeVertex(){
     int count = 0;
     int j = 0;
     int result;
     while(j < N){
       if((a[j].neighbourCount)>count){
        result=j;
        count = a[j].neighbourCount;
     }
     j++;
   }
   return result;
}

 Vertex GiveVetexInfo(int j){
    return a[j];
 }

};
```

```
struct CoverNode{
   int Size;
   int j;
   int* x;
   CoverNode(){
      x = new int[N-1];
   }
};

void Print(Vertex* v){
   int j = 0;
   cout << " Nebs" << endl;
   while(j < v->neighbourCount){

      cout << "  " << v->Adjacent[j];
      j++;
   }
}

void Print(CoverNode* Answer){
   cout << "\n answer=";
   cout << Answer->Size;
   cout << endl;
   /*
   int j = 0;
         int m = Answer->Size;
            while(j < m){
             //   cout << "  " << Answer->x[j] << "  ";
                j++;
            }
            */

}
class Stack{
   CoverNode*  y;
   int Top;
   public:
      Stack(){
         y = new CoverNode[N+1];
         Top = -1;
      }
   void Push(CoverNode& e){
      Top++;
      y[Top] = e;
   }
   CoverNode Pop(){
      Top--;
      return y[Top+1];
   }
   int StackEmpty(){
      if(Top==-1)
         return 1;
      return 0;
   }
};

int Search(int* a, int n, int k){
   int j = 0;
   while(j < n){
      if(a[j]==k)
      return 1;
      j++;
   }
   return -1;
}
//   Stack C;
void Apply(CoverNode& q, int i, Vertex* b, Stack&
C)
{
   q.j = i;
   int* a = b[i].Adjacent;
   int ncount = b[i].neighbourCount;
   int* d = q.x;
   int size = q.Size;
   int s =0;
   int j = 0;
   int e;
   while(j < ncount){
      if(Search(d,size,a[j])==1)
      s++;
      else{
         e = a[j];
      }
      j++;
   }
   if(s==ncount){

      return;
   }
   d[size]= b[i].vertexId;
   q.Size = q.Size +1;
   if(s<ncount-1)
      return;
      CoverNode z;
      z.Size = q.Size;
      z.j = i;
      int J = 0;
     while(J <z.Size){
```

```
            z.x[J] = q.x[J];
            J++;
        }
        z.x[z.Size-1] = e;

        C.Push(z);
}

int VertexCover(const Graph& a){

    Vertex* b = new Vertex[N];

    Graph temp = a;

    int j = 0; int odd = 1;
    while(temp.NodeCount()!=0){
        if(odd==1){

        int id = temp.MaxDegreeVertex();

        b[j]= temp.GiveVetexInfo(id);

        j++;
        temp.RemoveVertex(id);
        odd = 0;
    }
    else{
        int id = b[j-1].Adjacent[0];
        b[j]= temp.GiveVetexInfo(id);
        j++;
        temp.RemoveVertex(id);
        odd = 1;



    }
    }

CoverNode p;
p.Size = 0;
p.j = j;
Stack C;
C.Push(p);
CoverNode q;
int Min = N;
int counter=0;
while(C.StackEmpty()!=1) {
            if(counter==10)
// Increase the value of counter to get better result
            break;
            counter++;
             q = C.Pop();
  (q.j)--;
  int j = q.j;
   while(j >=0 ){
          if(q.Size>=Min)
          break;
          int ee = Min - q.Size;
          if((j/2)>ee)
          break;
          Apply(q,j,b,C);
          j--;

    }
    if(q.Size < Min){
       if(j==-1){
       p = q;
       Print(&p);
       Min = q.Size;
       }
    }
}
Print(&p);
  }

/* run this program using the console pauser or add
your own getch, system("pause") or input loop */

int main(int argc, char** argv) {

    ifstream File;
    File.open("graph450.txt");
    File >> N;
    cout <<"total number of nodes=" <<N;
    Graph a(File);
    VertexCover(a);
    File.close();
      return 0;
}
```