An Interesting Algorithm to Solve Vertex Cover Problem

D. Abhyankar, P. Saxena

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

Abstract: **- Vertex cover problem has been proved NP complete. Although a lot of research work has been carried out to invent approximation algorithms, exact algorithms to solve the problems remain unexplored to a large extent. We have found an algorithm that performs more efficiently than brute force search or simple backtracking algorithm. Our algorithm does not claim polynomial running time; however it solves the problem more efficiently than brute force search or simple backtracking algorithm.**

I. INTRODUCTION

A vertex cover or node cover of an input graph is the set of nodes such that each edge of the graph is covered by at least one vertex in the set. To find a minimum or optimum size cover is a classic problem of graph theory. To find optimum size vertex cover is a key problem in computational complexity theory. It is fixed parameter tractable [5].

Solutions to vertex cover problem can be applied in computational biochemistry and related areas. In biochemistry, sometimes we need to solve conflicts between sequences in a sample by ruling out some of the sequences. A conflict is strictly defined in the context of biochemistry. In a conflict graph the nodes or vertices epitomize the sequences in the sample. There is an edge between two vertices iff there is a conflict between the corresponding sequences. The aim is to remove the fewest possible vertices (sequences) that will resolve all conflicts. Recall that a vertex cover or node cover of an input graph is the set of nodes such that each edge of the graph is covered by at least one vertex in the set. Thus, the aim is to find a minimum vertex cover in the conflict graph G [6].

Vertex cover problem fascinates a lot of computer science researchers. In 1972, Researcher Karp proved this problem to be NP complete which means a polynomial time worst case solution to this problem is unlikely [1]. One way to tackle the problem is to look for approximation algorithms to solve this problem. For instance two-approximation algorithm solves the problem efficiently and elegantly. Also, literature describes some other important approximation algorithms. Another way is to solve the problem efficiently for special cases.

Although a lot of research work has been carried out in the area of approximation algorithms to solve vertex cover problem approximately, little efforts were made to find exact algorithms which are more efficient than brute force algorithms. Our study fills this gap and finds for exact algorithm which can perform more efficiently than brute force algorithm to solve vertex cover problem.

II. IDEAS BEHIND PROPOSED ALGORITHM

First idea of the algorithm is to leave single degree vertex. If we pick single degree vertex, we may end up in a cover with redundant vertex. A redundant vertex in a set S is one whose all neighbours are also present in set S. Covers with redundant vertex are sub optimal; therefore we should not compute covers with one or more redundant vertices.

Second idea of the algorithm is to compute close estimate of a graph. This close estimate helps the algorithm to achieve efficient backtracking. To have a close estimate we choose an edge and decide to have both ends in the estimate solution. Since at least one end must be picked by optimal solution, estimate <= 1+optimal. Since at least one end of the edge must be in the optimal solution, our recursion depth will not be deeper than twice of optimal depth.

III. PROPOSED ALGORITHM

We propose an algorithm that solves vertex cover problem more efficiently than brute force algorithms. Our algorithm involves three key ideas discussed in earlier Section. C++ code provided in the appendix implements the algorithm. Pseudocode of the algorithm has been presented below:

Function Exact(Graph G, int Lim) : int

Step 1: If Lim < 0 return N // Return value N indicates failure

Step 2: If there is a single degree vertex u, leave u. G0 = G – u – Neighbour(u)

Step 3: return Exact(G0, Lim-1)

Step 4: if Graph is edgeless return 0;

Step 5: Choose an edge (end1, end2). Choose both ends of the edge. G1 = G – end1 – end2.

Step 6: t = Exact(G1,Lim-1)

Step 7: If (t==N) return N // Backtracking

Step 8: CloseEstimate = t + 2

// Estimate provided by CloseEstimate can not be greater than 1+ExactAnswer, because at // least one end must be picked by exact solution.

Step 9: If (CloseEstimate < Lim) Lim = CloseEstimate

Step 10: Leave end1. G2 = G – end1 – Neighbours(end1)

Step 12: t = Exact(G2,Lim- NeighbourCount(end1))

Step 13: if (t !=N) return (t + NeighbourCount(end1))

Step 14: Leave end2. G3 = G – end2 – Neighbours(end2)

Step 15: t = Exact(G3,Lim- NeighbourCount(end2))

Step 16: if (t !=N) return (t + NeighbourCount(end2))

Step 17: Result = Min(CloseEstimate,LeaveEnd1,LeaveEnd2)

Step 18: return Result

IV. INFORMAL DESCRIPTION OF ALGORITHM

Function *Exact* computes the cardinality or size of minimum size vertex cover. Algorithm applies the ideas discussed in Section 3. *CloseEstimate*, *LeaveEnd1* and *LeaveEnd2* are key variables that store the intermediate results of the choices made by the algorithm. Variable Result stores the minimum of these key variables and forms the final answer which is returned by the algorithm in the last step.

V. CONCLUSION

A lot of work has been carried out in the field of approximation algorithms, but field of exact algorithms has been unexplored. To solve vertex cover problem, we can design much better than brute force algorithm or simple backtracking algorithm. Proposed algorithm combines three intuitive ideas and offers a better exact algorithm. Proposed algorithm does not claim polynomial running time, yet it solves the problem more efficiently than brute force search or simple backtracking algorithm.

REFERENCES

[1]. R.M. Karp, Reducibility among combinatorial problems, Complexity of Computer Computations, Plenum Press, 1972.
[2]. Stanley Lippman, Essential C++, Addison-Wesley, 2000.
[3]. 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.
[4]. http://www.dharwadker.org/vertex_cover/,2015
[5]. https://en.wikipedia.org/wiki/Vertex_cover,2015
[6]. http://www.dharwadker.org/pirzada/applications/2015

APPENDIX

```
using namespace std;
const int N = 7;

struct Vertex{
   int neighbourCount;
   int* Adjacent;
};

class Graph{
   private:
      int vertexCount;
      Vertex a[N];
   public:
      Graph(){
      vertexCount = N;
      int j = 0;
      while(j <N){
         cout << "Enter neighbour count of vertex "  << j  <<
endl;
         int count;
         cin >> count;
         a[j].neighbourCount = count;
         a[j].Adjacent = new int[count];
         int k = 0;
         while(k <count){
            cout << "Enter next neighbour-id" << endl;
            cin >> a[j].Adjacent[k];
            k++;
         }
         j++;

      }


   }

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

```
  int NodeCount() const{
     return vertexCount;
  }
  Graph(const Graph& ob){
     vertexCount = ob.vertexCount;
     int j = 0;
     while(j<N){
        a[j].neighbourCount = ob.a[j].neighbourCount;
        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++;

     }
  }

  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;
}

 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--;
   }

  }

  void RemoveAllNeighbour(int id){
       int count = a[id].neighbourCount;
       int j = 0;
       while(j < count){
          int tobeRemoved = a[id].Adjacent[j];
          RemoveVertex(tobeRemoved);
          j++;
       }
  }
  int GiveNeighbour(int id) const{
            return a[id].Adjacent [0];

   }

   int GiveNeighbour(int id1, int id2) const{
         int t = a[id1].Adjacent[0];
        if(a[id1].neighbourCount>1){
        if(t==id2)
          t = a[id1].Adjacent[1];
        }
        return t;
     }


  };
```

```
int VertexCover(const Graph& a);
int VertexCoverBackTrack(const Graph& a, int lim);

int  ChooseBothEnds(Graph& g,int end1, int end2){
   g.RemoveVertex(end1);
   g.RemoveVertex(end2);
   return 2+VertexCover(g);
}

int LeaveEnd(Graph& c,int lim, int end){
   int t = c.NebCount(end);
   lim = lim-t;
   c.RemoveAllNeighbour(end);
   int result= VertexCoverBackTrack(c,lim);
   if(result!=-1)
      result = result+t;
   return result;
}


int VertexCover(const Graph& a){
   if(a.NodeCount()>1){ // Graph is Non empty
      Graph b = a;
      int id = b.LeastDegreeVertex();
      if(b.NebCount(id)==1)
         {
            b.RemoveAllNeighbour(id);
             return 1+ VertexCover(b);
         }
      int end1 = b.GiveNeighbour(id);
      int end2 = b.GiveNeighbour(end1,id);
      int result1 = ChooseBothEnds(b,end1,end2);// Choose
both ends
      Graph c = a;
      int result2 = LeaveEnd(c,result1,end1); // Leave end1
      if(result2!=-1) return result2;
      Graph d = a;
      int result3 = LeaveEnd(d,result1,end2);    //Leave end2
      return result3;
   }
   else
   return 0; // Edgeless  Graph has cover of size 0
}
```

```
int VertexCoverBackTrack(const Graph& a, int lim){
   if(lim<0)
   return -1;  // No Graph can have cover of negative size so
function returns failure
   if(a.NodeCount()>1){ // Non Empty Graph
      Graph b = a;
      int id = b.LeastDegreeVertex();
      if(b.NebCount(id)==1)
         {
            b.RemoveAllNeighbour(id);
            int u =  VertexCoverBackTrack(b,lim-1);
             if(u!=-1)
             return u+1;
             return -1;


         }
      int end1 = b.GiveNeighbour(id);
      int end2 = b.GiveNeighbour(end1,id);
      int result1 = ChooseBothEnds(b,end1,end2); // Choose
both ends
      if(result1!=lim){
         if(result1<lim )
         return result1; // Answer found
         return -1;     // backtrack
      }
      Graph c = a;
      int result2 = LeaveEnd(c,result1,end1); // Leave end1
      if(result2!=-1){
            return result2;    // Answer found
      }
      Graph d = a;
      int result3 = LeaveEnd(d,result1,end2);    //Leave end2
      return result3;
   }
   else
   return 0; // Edgeless graph has cover size 0
}

int main(int argc, char** argv) {
   Graph x;
   cout << VertexCover(x);
   return 0;
}
```