Review Report on the

PACE 2016 Challenge Track B - Feedback Vertex Set

Implementation by Svyatoslav Feldsherov (Moscow State University)

Submitted By:

Sheshang M Ajwalia (M170096CS)

Navdeep Singh (M170262CS)

Vaibhav Lokhande (M170540CS)

Submitted to:

Dr. Subashini R.

at

National Institute of Technology, Calicut

Abstract:

In the mathematical discipline of graph theory, a *feedback vertex set* of a graph is a set of vertices whose removal leaves a graph without cycles. In other words, each feedback vertex set contains at least one vertex of any cycle in the graph. The feedback vertex set problem is an NP-complete problem in computational complexity theory. But this problem can be solved in polynomial time if we decide to measure with respect to the one parameter other than input size, and in feedback vertex set problem that parameter is the size of feedback vertex set.

Adopted Implementation:

We adopted the implementation which uses the Becker's algo for selecting random edge to go on next iteration. In addition to that it uses maximal independent set of multiply edges to recursively branch on both of the vertices of that edge.

Pseudo Code:

```
> ReadGraph(Cin, g)                          // Cin - input file , g - graph to be constructed

> MinFeedbackVertexSet(g, ans)               // ans : vector, storing the min feedback vertex set
  {
        n = Number of vertices in Graph

        for k=0 to n-2:
        deterministicBranchByMultiplyEdges( g, k, ans)

                makeRichGraph()                                   // doing kernalization
                        cutLeafs()                              // remove vertices with degree 1
                        cutLinkVertex()                         // remove vertices with degree 2
                        removeSelfLoop()          // remove self loops & add vertex to ans vector

        if ( k < 0 )
                ans.size = ans.size - DeleteSelfLoops
                return false

        If ( original_graph vertices == 0)
                return true

        If ( ! multiplyEdgesCount() )     // If there are no parallel edges between a pair of vertices
                if( !beckerguess() )                      // if BeckerGuess() didn't give vertex set
                        ans.size = ans.size - deleteSelfLoops
                        return beckerGuess;
```

```
        // Select any random multiply edge (b, e)
        g_without_b;
        G_without_e;

        // Recursively branch on both vertices of edge(b, e)

        if( deterministicBranchByMultiplyEdges(g_without_b, k, ans) )  return true;
        if( deterministicBranchByMultiplyEdges(g_without_e, k, ans) ) return true;
    }
> print ans;
```

Terminology:

Rich Graph:

A graph is called rich if every vertex is a branchpoint and it has no self-loops. Given a graph G, by repeatedly removing all leaves, and bypassing with an edge every linkpoint, a graph G' is obtained such that the size of a minimum FVS in G' and in G are equal and every minimum FVS of G' is a minimum FVS of G . Since every vertex involved in a self-loop belongs to every FVS, we can transform G' to a rich graph G(rich) by adding the vertices involved in self loops to the output of the algorithm.

BeckerGuess Algorithm :

In **Rich Graph**, take a random edge, select random vertex add it to the solution, recursively make it rich and call BeckerGuess() If at some iteration the graph is empty, all the vertices that we have chosen through this path is the right vertex set. If graph is not empty then remove the last added vertex from solution, take opposite vertex into consideration & continue BeckerGuess().

Complexity Analysis:

I is the total number of multiply edges in graph. We are selecting both of the vertices in two recursive branches, So total 2^I calls to deterministicBranchByMultiplyEdges(). In each of the call to deterministicBranchByMultiplyEdges(), BeckerGuess() is running for 4^K iterations. In each of the BeckerGuess(), itself is called recursively with decremented size of vertex set. The probability of success of BeckerSingleGuess() is $O(1/4^j)$, where j is the number of iterations in BeckerSingleGuess(). In this implementation, BeckerSingleGuess() is called upto I times, so here minimum probability of success of BeckerSingleGuess() is $O(1/4^I)$.
So final complexity = **O(** $2^I * 4^{(K-I)}$ **)**

Brute Force Approach:

Input: List of edges

Output: Feedback vertex set along with the execution time

Algorithm:

For each combination of the vertex set (in increasing order of size):
Reduce the original graph (by removing the edges which are adjacent to the vertices chosen in the current combination for fvs.)
Check Acyclicity of reduced graph with help of union-find algorithm.
If the reduced graph is acyclic
The current combination of vertex set is the minimum feedback vertex set.
Return that subset and terminate the program

Complexity Analysis:

Let say input has E edges and V vertices, Graph reading takes $O(E)$ time. Subset creation $O(2^V)$ time, as k-sub algorithm takes $O(N^K)$ time and V is for all the possibilities for the length of set. In *isAcyclic*() function, copy making of edgelist takes $O(E)$ time. Acyclicity checking takes $O(E * (logV)$ time (for each edge in edgelist copy, apply union find algorithm - both of which takes - $O(logV)$ and verify).
So total time complexity will be $O(2^V * E * log(V))$, as E will always be negligible in comparison to 2^V, asymptotic running time of FVS brute force approach is $O(2^V)$.

Comparisons:

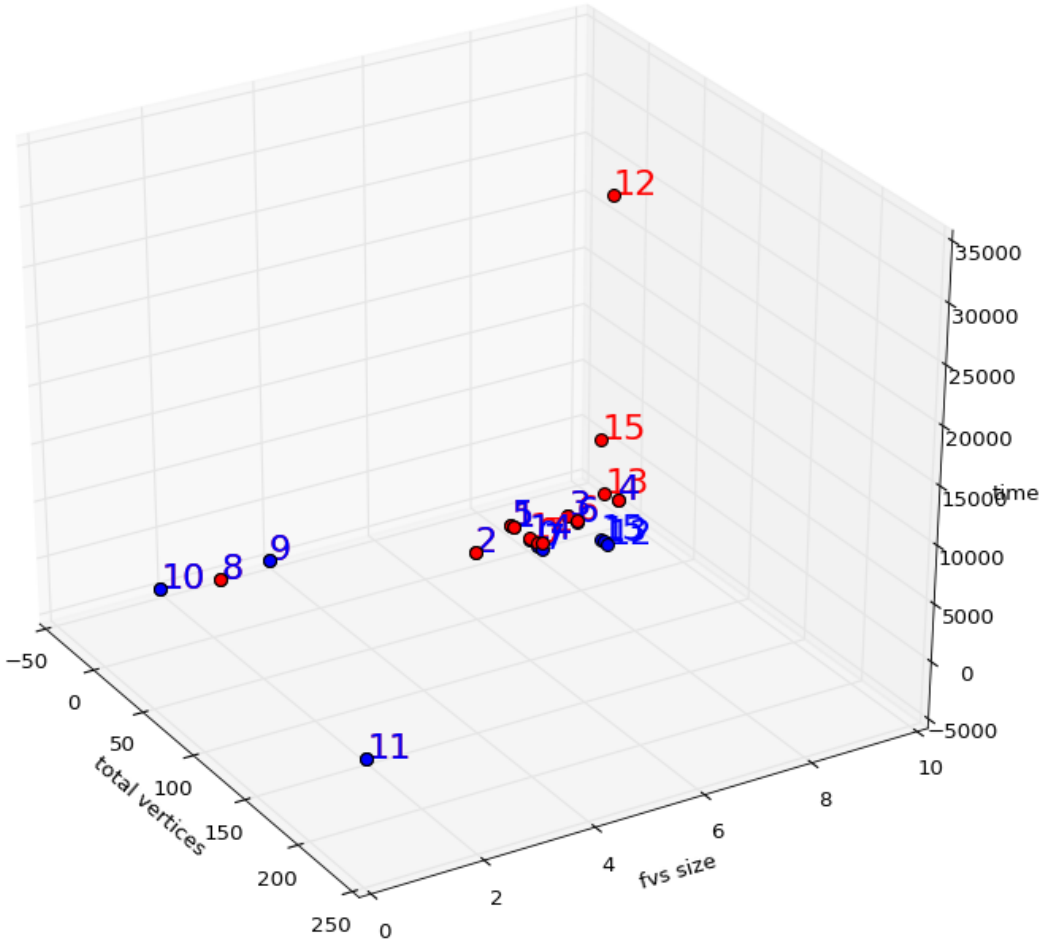
We have compared some of the public and hidden instances given on the challenge page on both of the implementations and here is the details about that.

Running Environment:

Memory: 3.8 GiB
Processor: Intel Core i5-4570 CPU @3.20GHz x 4

Graphical Comparison:

Here is the screenshot of the 3D plotting of total vertices in the instance **(n)**, size of fvs set **(k)** and running time **(t)** in seconds. Blue point shows parameterized running time while red shows brute force time. Annotation on the point shows the identifier for the instance, so that reader can compare the complexity of two version for same instance.



Tabular Comparison:

Graph Instance	V	E	FVS	BF RunTime	Parameterized RunTime
062.graph	57	78	7	227.72117	0.026048
083.graph	34	78	7	8.42384	0.08765
096.graph	48	64	6	15.87027	0.004836
099.graph	37	62	8	43.5356	0.05664
111.graph	36	76	9	132.22804	2.27915
119.graph	32	63	7	3.2978	0.042484
121.graph	45	64	8	136.12371	0.028042
84.graph	62	78	7	536.28449	0.018079
902.graph	9	9	2	0.00032	0.000276
903.graph	3	6	3	0.00014	0.000184
904.graph	3	3	1	0.00008	0.000204
906.graph	201	201	1	0.00054	0.002661
020.graph	74	92	8	30000.13622	0.001759
028.graph	70	85	8	4297.62917	9.60167
050.graph	49	84	7	92.8976	0.041539
065.graph	67	127	8	8694.17928	15.8216