TOPIC IN ALGORITHMS

FINAL REPORT

Feedback vertex set Problem

The Parameterized Algorithms and Computational Experiments Challenge

Pace 2016

Prepared by :

TGDK SUMANATHILAKA

B150413CS

sumanathilaka_b150413cs@gmail.com

Instructor : DR.SUBASHINI

COURSE : CS4027

DATE : 05 / 04 /2018

Table of Content

INTRODUCTION

The objective of this track is to solve the NP-hard Feedback Vertex Set Problem.

Input: An undirected graph.
Output: A minimum-size set of vertices such that deleting these vertices destroys all cycles (Forest)

(The program outputs a list of vertex names, one per line, of a smallest feedback vertex set of the input graph.)

In the *Feedback Vertex Set* problem algorithm given an undirected graph G and want to compute a smallest vertex set S such that removing S from G results in a forest, that is, a graph without any cycles. Feedback Vertex Set is NP-complete and one of the most prominent problems in parameterized algorithmics.

PREREQUISITES

Python 3. Tested against version 3.5.2.
python-igraph. Tested against version 0.7.1-5.

Installation

time python3 -O fvs.py <path to input-file>

Output

The program outputs a list of vertex names, one per line, of a smallest feedback vertex set of the input graph.

Definitions, Acronyms and Abbreviations

CLB	Current Lower Bound
FVS	Feedback Vertex Set
BFVS	Best Feedback Vertex Set

ALGORITHM

Overview

At a high level, the code is an implementation of the following simple algorithm, using the igraph library for graph operations:

1. Apply degree 0, 1, 2 reduction rules to get an equivalent graph of minimum degree at least 3.
2. Find a smallest cycle in the graph.
3. Branch on the vertices of this graph.

Details

The code implements some simple optimizations so that it doesn't (hopefully) take forever to run on moderately large instances. Here is a more detailed description of the code:

1. Read the **input file and construct a graph object**. Note that all graph operations use the igraph library.Igraph should be installed separately.
2. Apply the **degree-0, 1, 2 reduction rules** to get:
   - A reduced graph, possibly with multiple edges, whose minimum degree is at least 3. This graph does not contain loops, because every loop vertex is deleted from the graph and included in:
   - A partial feedback vertex set of the original graph, consisting of all those vertices which attain loop edges during the reduction process.
3. Compute **two lower bounds** on the size of an feedback vertex set of the reduced graph:
   - One bound is based on a degree argument, taking advantage of the fact that the reduced graph has minimum degree at least 3.
   - The other bound is based on greedily packing smallest cycles in the reduced graph, and counting the size of such a packing.
4. Add the size of the **partial fvs from step 2** to the higher of these two bounds, to obtain the "current lower bound" (CLB).
5. **Branch on the vertices of a shortest cycle** in the reduced graph, to find a smallest fvs of the reduced graph.
6. Output the list of vertices of the partial fvs from step 2 and those of the fvs obtained in step 5.

The branching algorithm

This algorithm branches on the vertex set of a shortest cycle of the reduced graph G. It employs some heuristics to try to bound the maximum branching depth. Following is a description of the main features of this algorithm:

1. The algorithm keeps track of the best fvs of G that we have found so far, at all times. In the beginning we apply a greedy approximation algorithm (pick the vertex with the largest degree, apply the reduction rules, and repeat) to get an approximation to the smallest fvs. This is our best fvs at this point.
2. We compute a CLB for the size of an fvs of G, exactly as in step 3 of the above algorithm. If the lower and upper bounds match, we have found a smallest fvs, and we return this fvs. Otherwise, we branch.
3. Our branching algorithm is as follows: We first find a smallest cycle in the graph. Let this cycle be v1, v2, ... , vp, v1.
    1. We find a smallest fvs F1 that contains the vertex v1. If the size of F1 matches the CLB for the graph, then we return this fvs. Otherwise we store F1 as our "best fvs" (BFVS) so far, and
    2. we find a smallest fvs F2 that *does not* contain v1, and contains v2. If the size of F2:
        - matches the CLB, then we return F2
        - is smaller than the size of F1, then we update our BFVS to F2
    3. If we did not return F2 in the previous step, then we find an fvs F3 that does *not* contain either of v1, v2, and contains v3. We then process F3 exactly as we did for F2.
    4. We keep doing this for successive vertices on the cycle, till we either:
        - Find an fvs F whose size matches our current lower bound, in which case we return F . or ,
        - Run out of vertices to branch on, in which case we return the then BFVS as a smallest fvs of the graph.
4. We use the following heuristics to speed up the branching:
    1. We do the branching "depth-first" rather than "breadth-first". That is, let G1 = G be the reduced graph with which we start, and let C1 be the first cycle of G1 that we branch on. We do not branch on all the vertices of C1 one after the other, as in the above description. Instead, we do the following:
    2. Let v1 be the first vertex of C1. We pick v1 into our solution, delete v1 from graph G1, and apply the reduction rules to the remaining graph to obtain graph G2.
    3. We then find a shortest cycle C2 of G2, pick the first vertex v2 of C2 into our solution, and apply the reduction rules to the remaining graph to obtain graph G3.
    4. We keep doing this till the remaining graph is empty.

5. Note that we go "deeper" into the branching tree first; that is, our branching picks vertices from disjoint cycles in preference to vertices from the same cycle. The intuitive reason to do this is that deleting vertices in this fashion is likely to result in a structurally simple (e.g: a disjoint collection of unicyclic graphs) graph sooner rather than later. This hunch has no theoretical basis (so far), but it seems to work well in practice.
6. We implement this "**depth-first"** branching by storing partially processed graphs (those we obtain by deleting a vertex from a shortest cycle) in a queue, and processing these partial solutions in queue order.
    1. Whenever it becomes clear that any fvs that we will find would not be smaller than the current best fvs that we have found, we stop our processing and abort the rest of that branch.
    2. Before branching on a new cycle, we check if our current partial solution is comparable in size to the current best fvs that we have. If this is indeed the case, then we find a lower bound for the fvs size of the remaining graph, and check if size(current partial solution) + lower_bound(remaining graph) is no smaller than the size of our current best fvs. If this is the case, then we abort this branch.

Implementation

Present Working PC

Properties :

Intel Core i3 1.90 GHz
4 GB Ram
500 GB Hard Drive

Operating System :

Ubuntu 17.04

Software

Python 3
python-igraph

Work Plan

SI NO	Activity	Date
1.	Introduction to the assignment. Understanding the purpose of the assignment and procedure.	2017-12-20
2.	Reference to the pace 2016 implementations and taking a rough idea on the topic	2017-12-20 to 2017-12-25
3.	Select the Topic(code) from the given codes.	2017-12-28
4.	Understanding the code.	2017-12-30 To
5.	Start with the Report .	2017-12-30
6.	Reference to documents and related resources .	2017-12-30 To 2018-01-15
7.	Compiling the code and check for the compile time etc.	2018-01-01 to 2018-01-22
8.	Mapping the above data to a graph	2018-01-27
9.	Finalizing the report	2018-04-04
10.	Submit the final Report	2018-04-05

RESULTS

Test Cases were downloaded from official pace 2016 website.

1.Test cases (hidden folder)

No	Graph No	No of Vertices	No of edges	Compile Time	Parameter (k)
1	119.graph	32	63	0.614 s	7
2	121.graph	45	64	0.598 s	8
3	111.graph	36	76	0.742 s	9
4	84.graph	62	78	0.731 s	7
5	127.graph	61	78	0.703 s	7
6	114.graph	55	81	0.604 s	11
7	72.graph	58	87	0.604 s	15
8	125.graph	69	96	0.679 s	8
9	115.graph	73	95	0.642 s	10
10	124..graph	74	101	0.736 s	8
11	120.graph	90	103	0.606 s	7
12	99.graph	59	104	6m 4.370 s	16
13	73.graph	70	105	0.604 s	18
14	74.graph	70	105	0.614 s	18
15	75.graph	70	105	0.609 s	18
16	2.graph	49	107	6.084 s	15

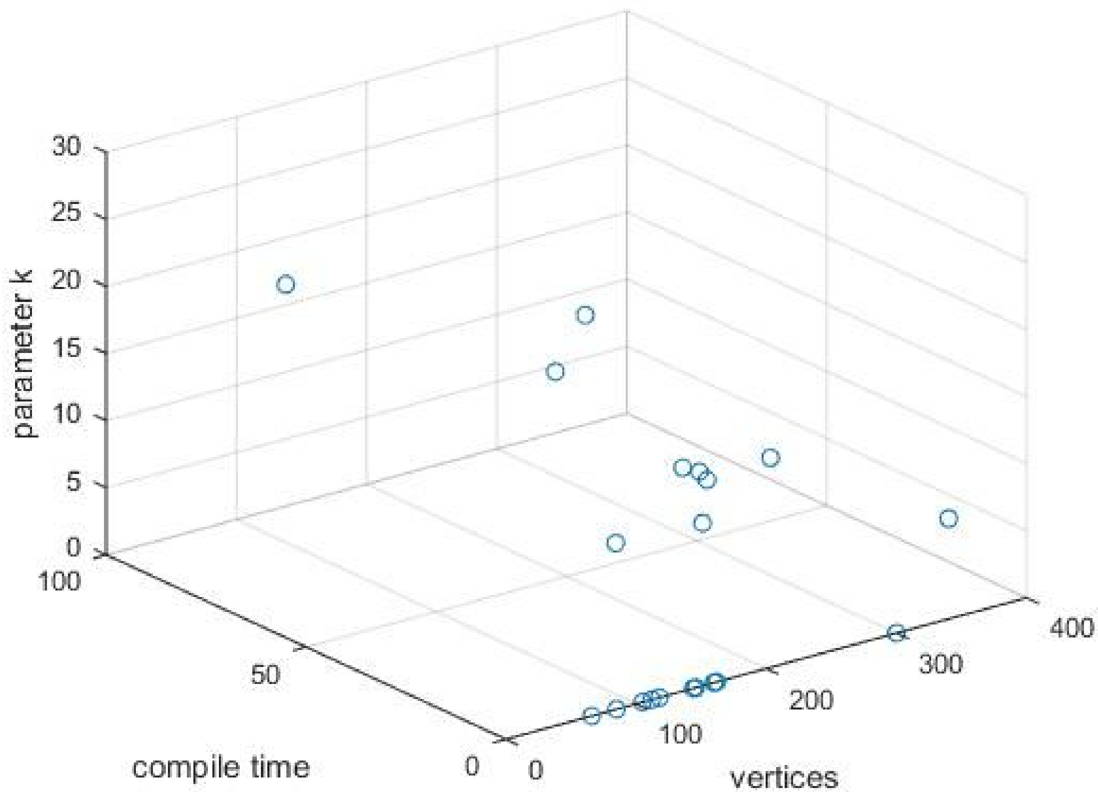
17	123.graph	71	115	0.725 s	11
18	56.graph	65	125	71 m 8.212 s (Out of time)	19
19	77.graph	76	152	5m 9.668 s	26
20	86.graph	62	159	3m 41.805 s	19
21	117.graph	125	146	1.3505 s	9
22	17.graph	80	167	1.1793 s	7
23	57.graph	112	168	1.1093 s	29
24	85.graph	39	170	2.3605 s	12
25	112.graph	153	177	0.6225 s	12
26	122.graph	145	186	2.996 s	16
27	126.graph	158	189	1.119 s	15
28	109.graph	66	192	Out of time	-
29	113.graph	149	193	0.0993 s	16
30	15.graph	89	206	1m 24.05 s	20
31	68.graph	85	219	Out of time	-
32	18.graph	90	231	1.825 s	12
34	4.graph	212	244	2.820 s	15
35	83.graph	61	248	31m 35.09s (Out of time)	30
36	37.graph	347	353	2.199 s	7
37	41.graph	146	361	Out of time	-
38	19.graph	100	391	20.164 s	22
39	12.graph	300	409	Out of time	-

40	1.graph	112	425	Out of time	-
41	106.graph	105	441	out of time	-
42	78.graph	118	531	Out of time	-
43	87.graph	162	510	Out of time	-
44	81.graph	144	576	Out of time	-
45	21.graph	160	620	Out of time	-

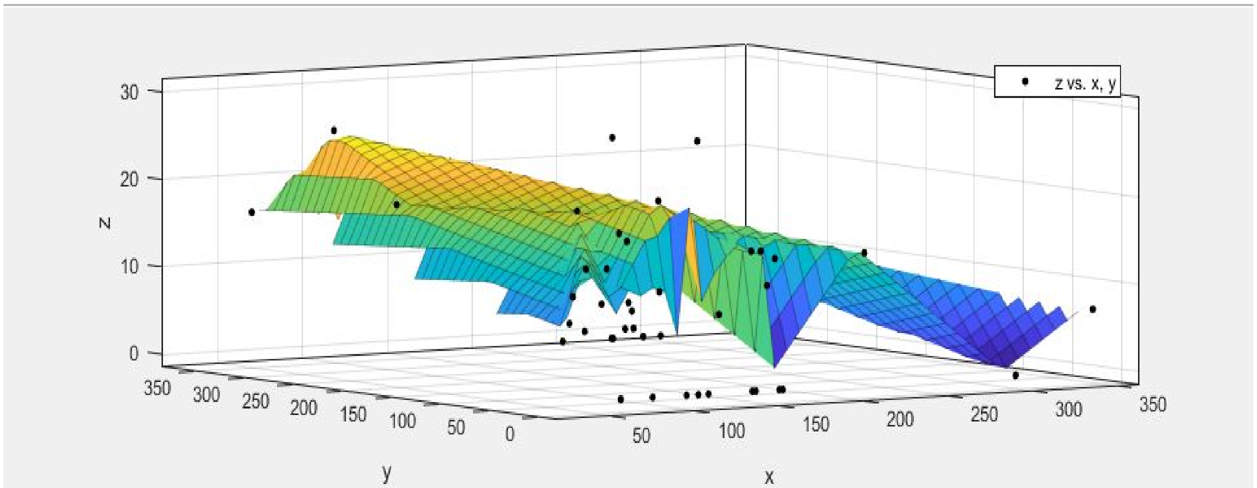
Summary:
No of test Cases : 130
No of test Cases pass with in 30 min of compile time : 33

2.Graph

Graphs are drawn using the **Matlab**
Input will be : No of vertices
Running time
Parameter (k)



The Point in the graph shows the variation of running time with respect to vertices and parameter k.
X axis = vertices
Y axis = compile time
Z axis = Parameter (k)



X = vertices of given graph
Y = Compile time (s)
Z = parameter (k)

This graph provides a graphical representation on the distribution of points with respect to X = vertices of given graph , Y = Compile time (s) , Z = parameter (k).

3. Test cases (Public Folder)

This test cases were given to the user to check for the correctness of the code . 100 instances were given.
In this assignment, smallest 45 instances was compiled and checked.

NO	Graph No	No of vertices	No of Edges	Compile Time	Parameter (K)
1	099.graph	37	62	0.608 s	8
2	096.graph	48	64	0.618 s	6
3	083.graph	34	78	0.618 s	7
4	062.graph	57	78	0.638 s	7
5	095.graph	34	83	0.686 s	8
6	050.graph	49	84	0.604 s	7

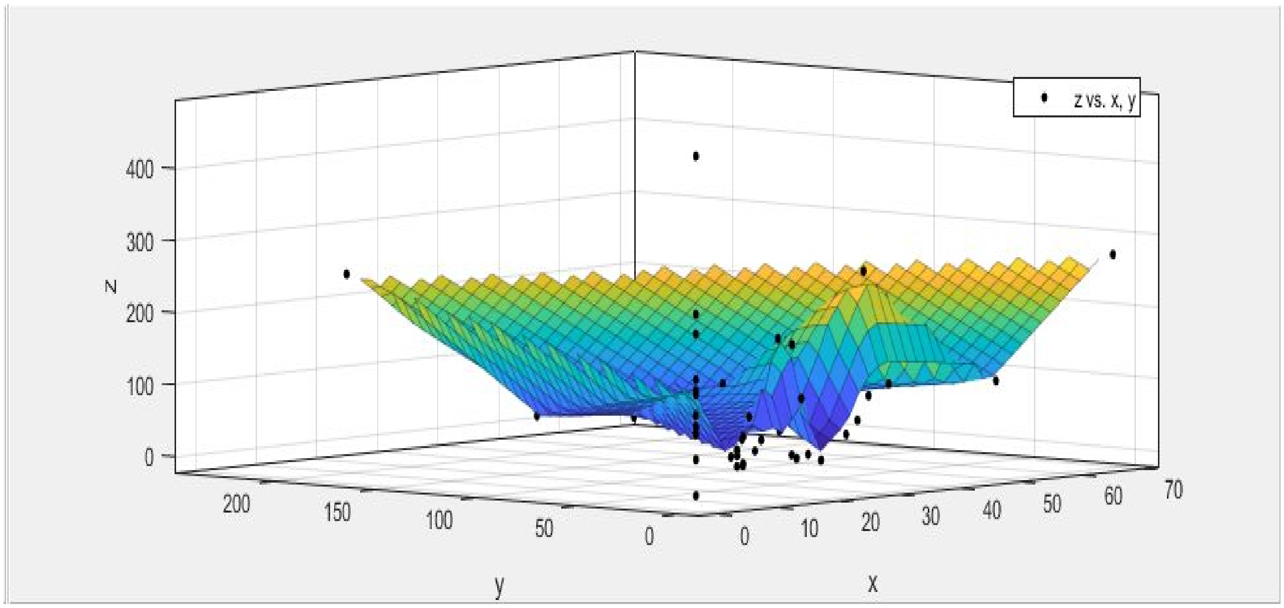
7	028.graph	70	85	1.166 s	8
8	003.graph	53	89	0.738 s	10
9	020.graph	74	92	0.601 s	8
10	042.graph	67	95	0.699 s	11
11	092.graph	42	105	0.615 s	16
12	065.graph	66	127	21.799 s	21
13	046.graph	73	152	1m 24.961 s	18
14	005.graph	62	159	2m 16.151 s	19
15	077.graph	113	161	8.557 s	16
16	029.graph	80	162	1.092 s	27
17	047.graph	84	166	Out of time	-
18	012.graph	112	168	1.637 s	29
19	051.graph	50	175	Out of time	-
20	015.graph	118	179	1.708 s	18
21	098.graph	118	179	1.938 s	18
22	060.graph	62	186	0.670 s	25
23	027.graph	126	189	0.625 s	32
24	072.graph	101	190	0.641 s	9
25	076.graph	87	227	Out of time	-
26	007.graph	36	239	1.192 s	17
27	070.graph	197	243	16.436 s	19
28	030.graph	40	292	1.442 s	19
29	091.graph	234	300	3m 56.627 s	21
30	024.graph	30	315	1.132 s	21
31	097.graph	96	336	Out of time	-

32	009.graph	110	364	24.675 s	21
33	061.graph	94	371	Out of time	-
34	059.graph	192	379	6.521 s	18
35	031.graph	278	394	16.138	33
36	086.graph	112	425	Out of time	-
37	026.graph	114	456	1.544s	50
38	013.graph	272	408	0.714	69
39	016.graph	224	420	Out of time	-
40	044.graph	120	469	58.934s	24
41	075.graph	252	476	Out of time	-
42	006.graph	471	503	Out of time	-
43	094.graph	161	608	Out of time	-
44	066.graph	146	657	Out of time	-
45	011.graph	140	698	Out of time	-

No of Tested Cases : 45
No of cases compiled with in 30 min : 33

4. Graph

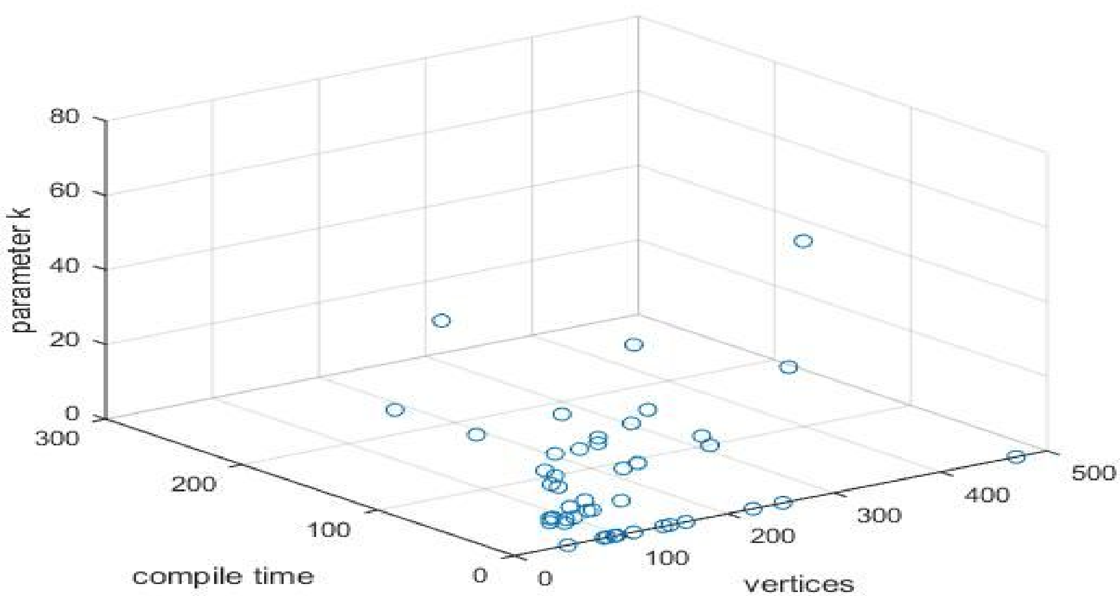
X = vertices of given graph
Y = Compile time (s)
Z = parameter (k)



Graphs are drawn using the **Matlab**
Input will be : No of vertices
Running time
Parameter (k)

The Point in the graph shows the variation of running time with respect to vertices and parameter k.

X axis = vertices
Y axis = compile time
Z axis = Parameter (k)



Summary

Kernelization techniques used in algorithm

1. If there is a loop in a vertex v, the vertex v will be delete from the graph.
   ({ G-v},{k-1})
2. If degree of a vertex is 2, then connects its neighbours and delete the vertex v .
   ({ G-v} , {k})
3. If degree of vertex v is less than or equal to 1 then delete that vertex v from the graph **({ G-v} , {k})**
4. If multiplicity of an edge 'e' is more than two , then it assign two as the multiplicity of e.
   ({ G-e} , {k})
5. Compute 2 Lower bounds using 2 different techniques
   a)degree argument - min degree 3
   b)Greedily packing smallest cycles
6. Compute Current Lower Bound.
7. Branch on the vertices of a shortest cycle in the reduced graph.

Complexity Discussion

In the (undirected) Feedback Vertex Set problem we are given an undirected graph G and want to compute a smallest vertex set S such that removing S from G results in a forest, that is, a graph without cycles. Feedback Vertex Set is NP-complete and one of the most prominent problems in parameterized algorithmics. Most fixed-parameter algorithms use the parameter solution size $k = | S |$.

Virtually all fixed-parameter algorithms make use of the fact that vertices of degree at most two can be easily removed from the graph. After this initial removal, a range of different techniques were used in the fixed-parameter algorithms. The first constructive **fixed-parameter algorithm branches on a shortest cycle in the resulting graph**. This cycle has length at most **2k** in a yes-instance, which results in an overall running time of **(2k)^k n^O(1).**

By using a **randomized approach** on the resulting graph, a running time of **4^k n^O(1)** can be obtained. The first deterministic approaches to achieve running times of the form **2^O(k).n^O(1)** use the **iterative compression technique.** It iteratively builds up the graph by adding one vertex at a time, and makes use of the fact that a size-k solution can be stored during this computation. Other fixed-parameter algorithms for this problem can be obtained by branching on a vertex of maximum degree or by LP-based techniques.

Brute Force Algorithm for Feedback Vertex Cover.

The Optimization problem of vertex cover was implemented.

The Compile time of every instance given was above 30 minutes.

With the above working conditions of the PC, Brute Force Algorithm came to end with a failure due to any test-case was unable to give a output within the time.

Conclusion

We can conclude that parameterized approach of FVS is far better than the Brute force approach of FVS .

Running time of the algorithm and complexity of the algorithm can be optimized using the parameterized approach.

References

1. Parameterized Algorithms and Computational Experiments Challenge
https://pacechallenge.wordpress.com/

2 .Google Scholar
https://scholar.google.co.in/scholar?q=feedback+vertex+cover&hl=en&as_sdt=0&as_vis=1&oi=scholart&sa=X&ved=0ahUKEwiF4q2xpMrYAhVJvY8KHQ2kCiQQgQMIJzAA

3. New Algorithms for k-Face Cover, k-Feedback Vertex Set, and k-Disjoint Cycles on Plane and Planar Graphs
https://link.springer.com/chapter/10.1007/3-540-36379-3_25

4. Approximations Algorithms by R.Ravi
https://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwigvvbyz9bYAhVGOI8KHY_iApMQFggmMAA&url=https%3A%2F%2Fwww.cs.cmu.edu%2Fafs%2Fcs%2Facademic%2Fclass%2F15854-f05%2Fwww%2Fscribe%2Flec9.ps&usg=AOvVaw3-MnhpubrC63tL3e2azwa2