Project Documentation

Danny Elliott, November 25, 2017

A. Hardware and Software Environment

This project was developed on a 2011 MacBook Pro with 16 GB of RAM and a 2.4 GHz Intel Core i5. The assignment was coded in Java using the IntelliJ IDEA CE integrated development environment.

B. Algorithm Design

I chose to find solutions to the Vertex-Cover Problem.
My initial data setup consisted of the following:

- A Graph object which contained an array of n Edge objects and an array of n CoverageBreakdown objects
- Each Edge would have a vertexA and vertexB attribute, both represented as integers. Vertices in the graph are represented from 0 to n-1 vertices.
- Each CoverageBreakdown would contain an integer array where the first element is the "leading" vertex of an edge, and all other elements are vertices on edges incident to the leading vertex. The CoverageBreakdown also has a firstVertex attribute which stores the leading vertex of the edge. This is needed since the greedy algorithm is coded in such a way that the coverage arrays are altered, yet the first vertex still needs to be referenced.

As an example: A graph G contains an array of edges in the form of vertexA => vertexB

[0 => 1
1 => 3
2 => 4
3 => 0
4 => 1]

and an array of coverage breakdowns where the first vertex of each breakdown gets saved in the firstVertex attribute.

[[0, 1, 3]
[1, 3, 0, 4]
[2, 4]
[3, 0, 1]
[4, 1, 2]]

Brute-Force Algorithm

I started with a brute-force approach where I attempted to find the best covering for n edges. I would randomly shuffle the order of the edges before running the algorithm through n^(n/2) iterations. Though this isn't guaranteed to run through every possible combination of edge orders, it does use the same amount of time as an algorithm that would hit every combination.

	ITERATIVE-BRUTE-FORCE-VERTEX-COVER (Graph G)	
0	Let n = number of edges in G	(1)
1	bestSize = n // start with the assumption that each edge must be selected	(1)
2	let edgeList, tempBestCover, and bestCover be new empty arrays	(1)
3	for k = 1 to n^(n/2)	$n^{(n/2)}+1$
4	for i = 1 to n	$n^{(n/2)+1}+1$
5	*// scan coverage breakdown for each edge i*	
6	currentBreakdownArray = G.coverageBreakdown[i]	$n^{(n/2)+1}$
7	for x = 1 to currentBreakdownArray.length	$n^{(n/2)+2}+1$
8	currentVertex = currentBreakdownArray[x]	$n^{(n/2)+2}$
9	if edgeList does NOT contain currentVertex	$n^{(n/2)+2}$
10	add currentVertex to edgeList	$n^{(n/2)+2}$
11	*// first vertex represents current edge*	
12	firstVertex = currentBreakdownArray[1]	$n^{(n/2)+2}$
13	if tempBestCover does NOT contain firstVertex	$n^{(n/2)+2}$
14	add firstVertex to tempBestCover	$n^{(n/2)+2}$
15	if tempBestCover.length < bestSize	$n^{(n/2)+1}$
16	bestSize = tempBestCover.length	$n^{(n/2)+1}$
17	bestCover = tempBestCover	$n^{(n/2)+1}$
18	shuffle order of arrays in G.coverageBreakdown	$n^{(n/2)+1}$
19	empty tempBestCover	$n^{(n/2)+1}$
20	empty edgeList	$n^{(n/2)+1}$
21	return bestCover	(1)

This algorithm runs in $O(n^n)$ since the running time is dominated by n^n.

Greedy Algorithm

In an attempt to speed up the running time, I implemented an algorithm where the coverage arrays would be chosen greedily by the number of vertices that had not already been picked.

	GREEDY-VERTEX-COVER (Graph G)	
0	Let n = number of edges in G	(1)
1	let edgeList, and bestCover be new empty arrays	(1)
2	sort arrays in G.coverageBreakdown by descending length	n^2
3	while edgeList.length < n	$(\sum 0 \rightarrow n)$
4	currentBreakdownArray = G.coverageBreakdown[1]	$(\sum 0 \rightarrow n)$
5	for k = 1 to currentBreakdownArray.length	$(\sum 0 \rightarrow n)$ k+1
6	currentVertex = currentBreakdownArray[k]	$(\sum 0 \rightarrow n)$ k
7	if edgeList does NOT contain currentVertex	$(\sum 0 \rightarrow n)$ k
8	add currentVertex to edgeList	$(\sum 0 \rightarrow n)$ k
9	*// first vertex represents current edge*	
10	firstVertex = currentBreakdownArray[1]	$(\sum 0 \rightarrow n)$ k
11	if bestCover does NOT contain firstVertex	$(\sum 0 \rightarrow n)$ k
12	add firstVertex to bestCover	$(\sum 0 \rightarrow n)$ k
13	for x = 1 to edgeList.length	$(\sum 0 \rightarrow n)$ kx+1
14	currentBreakdownArray = G.coverageBreakdown[i]	$(\sum 0 \rightarrow n)$ kx
15	remove vertex in edgeList from currentBreakdownArray	$(\sum 0 \rightarrow n)$ kx
16	// sort after each pass of the current top coverage array	
17	sort arrays in G.coverageBreakdown by descending length	$(\sum 0 \rightarrow n)$ n^2
18	return bestCover	(1)

This greedy algorithm runs in $O(n^2)$ time.

Approximate Algorithm

Lastly, I implemented an approximate algorithm based on section 35.1 of the textbook. Due to it being an approximation, both vertices on a selected edge get returned in the bestCover array. This means that the number of vertices returned is double the minimum vertex cover. This is in keeping with APPROX-VERTEX-COVER being a p(2)-approximation algorithm.

* When analyzing the run time, let n be the current number of edges in G.

	APPROX-VERTEX-COVER (Graph G)	
0	let bestCover be a new empty array	(1)
1	while G.edges.length > 0	(∑0→n*)
2	firstEdge = G.edges[1]	(∑0→n)(1)
3	firstVertex = firstEdge.vertexA	(∑0→n)(1)
4	secondVertex = firstEdge.vertexB	(∑0→n)(1)
5	if bestCover does NOT contain firstVertex	(∑0→n)(1)
6	add firstVertex to bestCover	(∑0→n)(1)
7	if bestCover does NOT contain secondVertex	(∑0→n)(1)
8	add secondVertex to bestCover	(∑0→n)(1)
9	let coveredVerts be a new array = [firstVertex, secondVertex]	(∑0→n)(1)
10	//remove current and incident edges from G.edges	
11	for i = 1 to G.edges.length	(∑0→n)(n+1)
12	for x = 1 to 2	(∑0→n)(n)(3)
13	if G.edges[i].vertexA = coveredVerts[x] OR	(∑0→n)(2)
14	if G.edges[1].vertexB = coveredVerts[x]	(∑0→n)(2)
15	remove G.edges[i] from G.edges	
16		
17	return bestCover	(1)

The approximation algorithm runs in O(n) time

C. Testing

Tests were conducted by creating a random graph with n edges (where n is the first argument passed into the problem) and running the graph through the three algorithms. A Timer class was used to measure the run time (in milliseconds) of each vertex cover algorithm. The output of each algorithm was the array of vertices chosen for the vertex cover. A message is also shown indicating if a cover was found with the max number of vertices passed in. (Note that in the case of the approximation algorithm, more vertices show up since both vertices in a selected edge are added).

Test 1: 6 edges and a target coverage of 2 vertices.

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_25.jdk/Contents/Home/bin/java ...
Graph Edges:
0 => 5
1 => 3
2 => 1
3 => 4
4 => 2
5 => 4

Order of coverings:
[0, 5]
[1, 3, 2]
[2, 1, 4]
[3, 4, 1]
[4, 2, 3, 5]
[5, 4, 0]

Best Brute Force Coverage after 216.0 iterations: [1, 5]
Found in 30ms
Brute Force Algorithm was able to find coverage of 2 or less veritices

Best Greedy Coverage: [4, 0, 3]
Found in 16ms
Greedy Algorithm was not able to find coverage of 2 or less veritices

Best Approximate Coverage:[0, 5, 1, 3, 4, 2]
Found in 0ms
Approximation Algorithm was not able to find coverage of 2 or less veritices


Process finished with exit code 0
```

Test 2: 9 edges and a target coverage of 4 vertices.

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_25.jdk/Contents/Home/bin/java ...
Graph Edges:
0 => 8
1 => 4
2 => 4
3 => 7
4 => 6
5 => 6
6 => 2
7 => 6
8 => 1

Order of coverings:
[0, 8]
[1, 4, 8]
[2, 4, 6]
[3, 7]
[4, 6, 1, 2]
[5, 6]
[6, 2, 4, 5, 7]
[7, 6, 3]
[8, 1, 0]

Best Brute Force Coverage after 6561.0 iterations: [3, 8, 6]
Found in 92ms
Brute Force Algorithm was able to find coverage of 4 or less veritices

Best Greedy Coverage: [6, 8, 7]
Found in 2ms
Greedy Algorithm was able to find coverage of 4 or less veritices

Best Approximate Coverage:[0, 8, 1, 4, 3, 7, 5, 6]
Found in 1ms
Approximation Algorithm was able to find coverage of 4 or less veritices


Process finished with exit code 0
```

Test 3: 12 edges and a target coverage of 5 vertices.

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_25.jdk/Contents/Home/bin/java ...
Graph Edges:
0 => 1
1 => 8
2 => 5
3 => 5
4 => 2
5 => 1
6 => 0
7 => 9
8 => 4
9 => 3
10 => 8
11 => 7

Order of coverings:
[0, 1, 6]
[1, 8, 0, 5]
[2, 5, 4]
[3, 5, 9]
[4, 2, 8]
[5, 1, 2, 3]
[6, 0]
[7, 9, 11]
[8, 4, 1, 10]
[9, 3, 7]
[10, 8]
[11, 7]

Best Brute Force Coverage after 2985984.0 iterations: [5, 0, 8, 7]
Found in 6634ms
Brute Force Algorithm was able to find coverage of 5 or less veritices

Best Greedy Coverage: [8, 3, 0, 7, 4]
Found in 2ms
Greedy Algorithm was able to find coverage of 5 or less veritices

Best Approximate Coverage:[0, 1, 2, 5, 7, 9, 8, 4]
Found in 0ms
Approximation Algorithm was able to find coverage of 5 or less veritices


Process finished with exit code 0
```

D. Running the Program

The program accepts two parameters, one for the number of edges in the graph and one for maximum target vertex cover. They are entered in the form…

```
java VertexCover edgeCount coverTarget
java VertexCover 11 4
```

You will find the code in the Vertex-Cover folder; navigate to that folder in a terminal before running the program.

E. Knowledge Gained

I found this project challenging since it had been a while since I've used Java. I enjoyed getting back into it with this project. Running the finished code confirmed my suspicions that the brute-force algorithm would return the best results but at a great cost; running n^(n/2) iterations quickly increases the run time as n increases. On my machine, I could only run a test with 12 or 13 iterations before the program would become unresponsive.

The code for the greedy algorithm shares some similarities to the brute-force approach. However, it removed the large number or iterations and always picked the cover grouping with the largest number of remaining vertices while culling visited vertices on each iteration. I was surprise to see that the greedy algorithm didn't always return the smallest vertex cover.

As I suspected from the material in chapter 35.1, the approximation algorithm was very fast. Since two vertices got added to the vertex cover each time an edge was removed, I was unsure if each edge still counted as one vertex in the cover. Since it is a p(2)-approximation algorithm, I'm guessing not. This would mean that even though the approximation algorithm executes extremely fast, it will return double the number of vertices in the cover. This could be viewed as the approximation trade-off.

F. References

T. Cormen, Introduction to Algorithms: Third Edition, The MIT Press, Cambridge, Massachusetts, 2009.