Charles Sedgwick

Exploring Methods to Find an Approximately Minimal Vertex Cover in Polynomial Time

Algorithm Design

First, a brute force algorithm to solve the vertex cover problem was implemented. This algorithm generates all subsets starting with size 1 to |G.V|-1 and for each subset generates checks if it is a vertex cover of the graph. In order to implement this algorithm, a function is required that given a set of edges and a possible cover for a graph returns true if the cover is a vertex cover or false otherwise. Also required is a function that, given a graph and a size k, returns all subsets size k subsets of the graphs vertices. The vertex_cover_brute(graph,res) function takes a graph with k vertices and for each value n in range [1,k] generates the n size subsets of the vertices. For each n size subset, it checks if it is a vertex cover of the graph. If so then it is stored in res and the function returns. One optimization applied to this brute force algorithm was to generate the subsets in increasing size so that as soon as a vertex cover is found it can be taken as the minimal vertex cover. This does not improve the worst case time beyond a constant factor, however, as the algorithm may still need to generate all subsets with sizes in the range [1,k] before finding a cover.

In order to improve on the exponential time of the brute force algorithm described here, an approximation algorithm was designed that took advantage of the fact that the higher out degree of a vertex in the graph, the more likely it is to be in the minimal vertex cover. The vertex_cover_degree(graph, res) function takes a graph and first generates a dictionary of (vertex, degree) for each vertex in the graph. A dictionary in Python is also known as an associative array. They are not indexed by a range of values but rather by a set of immutable keys such as strings. This dictionary is then sorted so the vertices with the highest degree are near the front of the dictionary. Then, after an empty cover is created, a vertex is added to the cover and the cover is checked to see if it is a minimal vertex cover for the graph. If not, then the next vertex in the dictionary is added. This process continues until a minimal vertex cover is found. To help determine how this algorithm performs, the 2-approximation algorithm APPROX_VERTEX_COVER(G) from Cormen et al. was also implemented (Cormen, 2009).

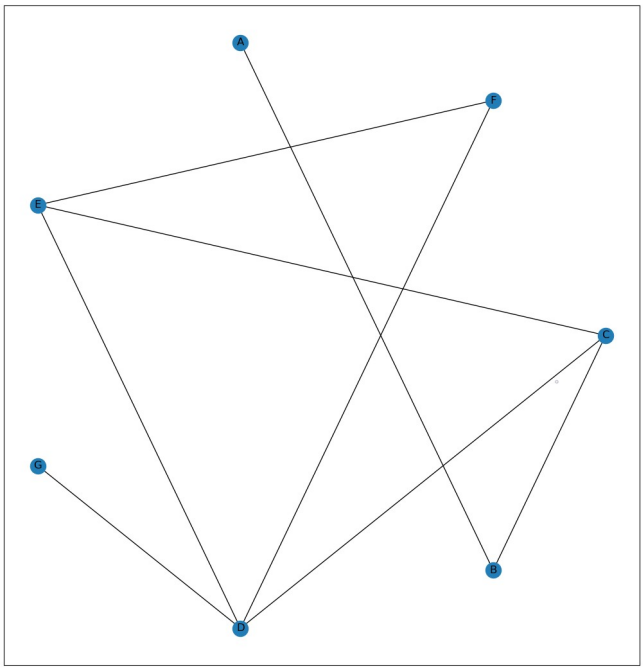
Results

In implementing the algorithms described above for solving the vertex cover problem, the following software and hardware was used:

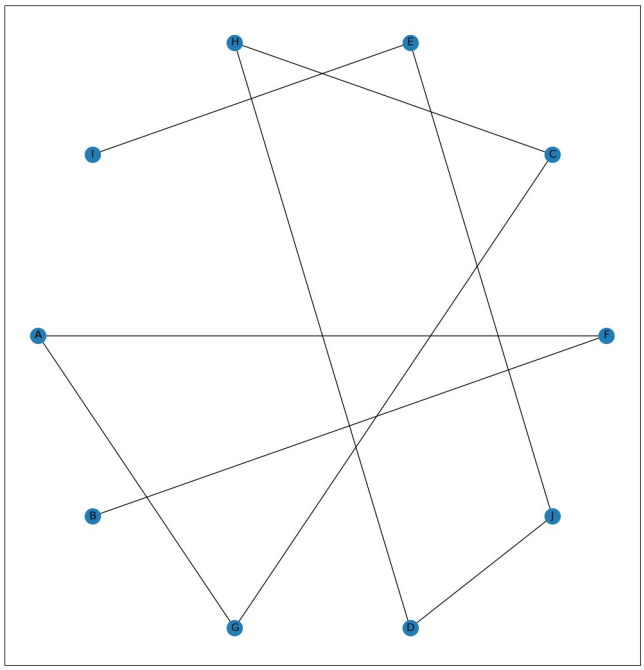
- Operating System: Windows 10
- Programming Language: Python 3.5
  - IDE: PyCharm 2020.1
    - no specific features of PyCharm are required to run the included script
- Third party libraries:
  - matplotlib and networkx for plotting graphs and results table
  - timeit for timing execution time of algorithms
- CPU: AMD Ryzen 2600X 6 core processor
- Memory: 32 GB DDR4
- storage: SSD

Charles Sedgwick

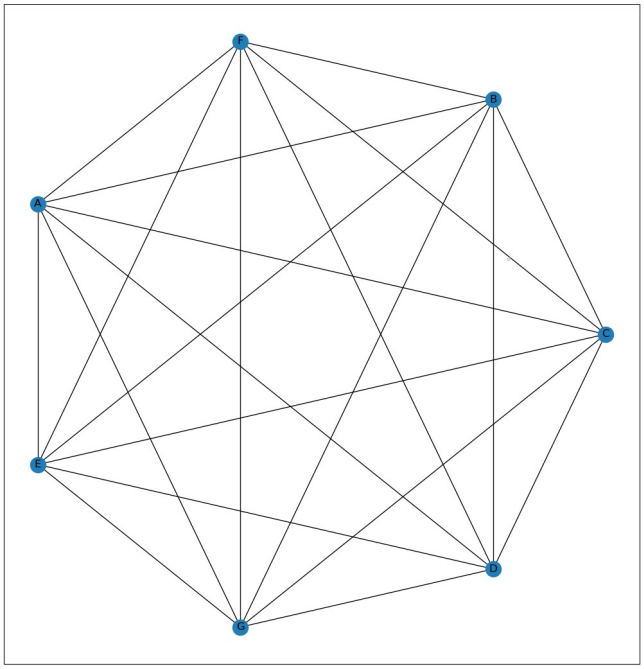
The following graphs were used to test the algorithms described:



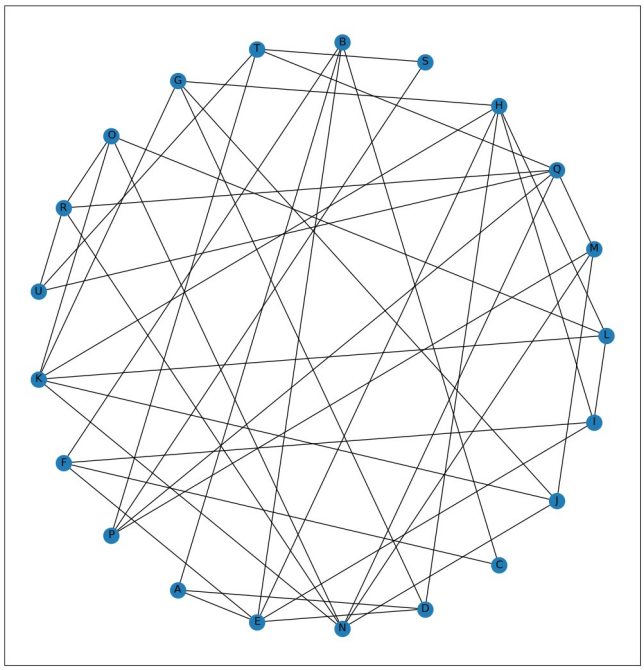
Graph351



GraphBipartite



GraphConnected



GraphBig

Using the graphs above, the following results were recorded for 6 trials:

Graph	Algorithm	Running Time (ms)	Approximation Ratio	Cover
Graph351	Brute Force	0.219	1.0	['B', 'D', 'E']
Graph351	Approximation	0.025	1.333	['B', 'C', 'D', 'E']
Graph351	Degree Heuristic	0.038	1.333	['D', 'C', 'E', 'B']
GraphConnected	Brute Force	1.473	1.0	['B', 'D', 'C', 'E', 'G', 'F']
GraphConnected	Approximation	0.032	1.000	['B', 'A', 'D', 'C', 'E', 'F']
GraphConnected	Degree Heuristic	0.101	1.000	['B', 'D', 'C', 'E', 'G', 'F']
GraphBipartite	Brute Force	2.815	1.0	['B', 'D', 'C', 'E', 'A']
GraphBipartite	Approximation	0.016	1.600	['B', 'F', 'D', 'H', 'J', 'E', 'C', 'G']
GraphBipartite	Degree Heuristic	0.039	1.200	['D', 'H', 'C', 'E', 'F', 'G']
GraphBig	Brute Force	31470.805	1.0	['N', 'T', 'J', 'H', 'B', 'D', 'F', 'R', 'L', 'P', 'E', 'Q', 'K']
GraphBig	Approximation	0.098	1.385	['B', 'A', 'D', 'E', 'M', 'J', 'N', 'K', 'C', 'F', 'T', 'S', 'G', 'H', 'R', 'Q', 'L', 'I']
GraphBig	Degree Heuristic	0.470	1.231	['N', 'K', 'H', 'E', 'Q', 'B', 'J', 'M', 'T', 'F', 'G', 'D', 'L', 'O', 'R', 'P']

Table 1: Trial 1

Graph	Algorithm	Running Time (ms)	Approximation Ratio	Cover
Graph351	Brute Force	0.338	1.0	['D', 'E', 'B']
Graph351	Approximation	0.024	1.333	['C', 'B', 'F', 'D']
Graph351	Degree Heuristic	0.046	1.667	['D', 'C', 'E', 'F', 'B']
GraphConnected	Brute Force	1.639	1.0	['C', 'A', 'F', 'G', 'D', 'E']
GraphConnected	Approximation	0.040	1.000	['C', 'A', 'F', 'D', 'G', 'E']
GraphConnected	Degree Heuristic	0.106	1.000	['C', 'A', 'F', 'G', 'B', 'D']
GraphBipartite	Brute Force	3.873	1.0	['A', 'D', 'B', 'C', 'E']
GraphBipartite	Approximation	0.018	1.600	['C', 'H', 'J', 'D', 'A', 'F', 'E', 'I']
GraphBipartite	Degree Heuristic	0.068	1.600	['G', 'J', 'A', 'H', 'F', 'C', 'D', 'E']
GraphBig	Brute Force	34048.442	1.0	['K', 'H', 'Q', 'D', 'B', 'L', 'F', 'P', 'R', 'T', 'N', 'J', 'E']
GraphBig	Approximation	0.112	1.538	['P', 'M', 'K', 'J', 'A', 'B', 'H', 'G', 'R', 'Q', 'D', 'E', 'N', 'O', 'L', 'I', 'C', 'F', 'S', 'T']
GraphBig	Degree Heuristic	0.416	1.077	['K', 'H', 'Q', 'N', 'E', 'P', 'R', 'M', 'T', 'D', 'B', 'L', 'J', 'F']

Table 2: Trial 2

Graph	Algorithm	Running Time (ms)	Approximation Ratio	Cover
Graph351	Brute Force	0.248	1.0	['B', 'D', 'E']
Graph351	Approximation	0.019	1.333	['B', 'C', 'D', 'E']
Graph351	Degree Heuristic	0.044	1.333	['D', 'C', 'E', 'B']
GraphConnected	Brute Force	1.514	1.0	['B', 'D', 'G', 'C', 'A', 'F']
GraphConnected	Approximation	0.034	1.000	['B', 'A', 'D', 'C', 'G', 'E']
GraphConnected	Degree Heuristic	0.107	1.000	['B', 'D', 'G', 'C', 'A', 'F']
GraphBipartite	Brute Force	4.010	1.0	['H', 'G', 'F', 'D', 'E']
GraphBipartite	Approximation	0.017	2.000	['B', 'F', 'H', 'C', 'D', 'J', 'G', 'A', 'I', 'E']
GraphBipartite	Degree Heuristic	0.051	1.400	['H', 'D', 'G', 'C', 'F', 'A', 'E']
GraphBig	Brute Force	36044.503	1.0	['T', 'N', 'Q', 'L', 'B', 'F', 'E', 'J', 'K', 'R', 'H', 'D', 'P']
GraphBig	Approximation	0.110	1.385	['K', 'J', 'H', 'G', 'T', 'S', 'F', 'E', 'O', 'N', 'R', 'Q', 'L', 'I', 'B', 'A', 'P', 'M']
GraphBig	Degree Heuristic	0.640	1.231	['K', 'H', 'N', 'Q', 'E', 'T', 'G', 'R', 'L', 'B', 'D', 'O', 'F', 'I', 'P', 'M']

Table 3: Trial 3

Graph	Algorithm	Running Time (ms)	Approximation Ratio	Cover
Graph351	Brute Force	0.285	1.0	['D', 'B', 'E']
Graph351	Approximation	0.018	2.000	['D', 'C', 'B', 'A', 'F', 'E']
Graph351	Degree Heuristic	0.048	1.333	['D', 'E', 'C', 'B']
GraphConnected	Brute Force	1.519	1.0	['D', 'B', 'F', 'E', 'A', 'G']
GraphConnected	Approximation	0.039	1.000	['D', 'A', 'B', 'C', 'F', 'E']
GraphConnected	Degree Heuristic	0.125	1.000	['D', 'B', 'F', 'E', 'A', 'G']
GraphBipartite	Brute Force	3.396	1.0	['D', 'B', 'A', 'C', 'E']
GraphBipartite	Approximation	0.017	1.600	['D', 'H', 'B', 'F', 'E', 'I', 'A', 'G']
GraphBipartite	Degree Heuristic	0.071	1.400	['D', 'F', 'E', 'J', 'A', 'G', 'H']
GraphBig	Brute Force	33620.134	1.0	['T', 'E', 'R', 'P', 'D', 'B', 'F', 'Q', 'J', 'L', 'K', 'N', 'H']
GraphBig	Approximation	0.127	1.538	['D', 'A', 'B', 'E', 'F', 'C', 'Q', 'P', 'T', 'S', 'J', 'G', 'L', 'K', 'O', 'N', 'U', 'R', 'I', 'H']
GraphBig	Degree Heuristic	0.605	1.231	['Q', 'E', 'N', 'K', 'H', 'D', 'O', 'F', 'T', 'L', 'G', 'B', 'J', 'M', 'R', 'P']

Table 4: Trial 4

Graph	Algorithm	Running Time (ms)	Approximation Ratio	Cover
Graph351	Brute Force	0.417	1.0	['D', 'E', 'B']
Graph351	Approximation	0.023	2.000	['F', 'D', 'E', 'C', 'B', 'A']
Graph351	Degree Heuristic	0.063	1.667	['D', 'E', 'C', 'F', 'B']
GraphConnected	Brute Force	1.729	1.0	['F', 'D', 'E', 'G', 'C', 'A']
GraphConnected	Approximation	0.041	1.000	['F', 'A', 'D', 'C', 'E', 'B']
GraphConnected	Degree Heuristic	0.134	1.000	['F', 'D', 'E', 'G', 'B', 'A']
GraphBipartite	Brute Force	4.425	1.0	['I', 'H', 'F', 'J', 'G']
GraphBipartite	Approximation	0.018	1.600	['F', 'A', 'I', 'E', 'D', 'H', 'G', 'C']
GraphBipartite	Degree Heuristic	0.063	1.200	['F', 'D', 'E', 'J', 'G', 'C']
GraphBig	Brute Force	32969.937	1.0	['L', 'B', 'H', 'F', 'K', 'T', 'E', 'N', 'P', 'R', 'D', 'J', 'Q']
GraphBig	Approximation	0.141	1.538	['I', 'H', 'L', 'K', 'R', 'Q', 'D', 'A', 'M', 'J', 'O', 'N', 'B', 'E', 'S', 'P', 'F', 'C', 'T', 'U']
GraphBig	Degree Heuristic	0.698	1.308	['K', 'H', 'Q', 'E', 'N', 'L', 'I', 'D', 'M', 'O', 'B', 'F', 'J', 'R', 'T', 'G', 'P']

Table 5: Trial 5

Graph	Algorithm	Running Time (ms)	Approximation Ratio	Cover
Graph351	Brute Force	0.281	1.0	['D', 'E', 'B']
Graph351	Approximation	0.024	2.000	['D', 'C', 'F', 'E', 'B', 'A']
Graph351	Degree Heuristic	0.086	1.667	['D', 'C', 'E', 'F', 'B']
GraphConnected	Brute Force	1.318	1.0	['D', 'G', 'F', 'C', 'A', 'E']
GraphConnected	Approximation	0.048	1.000	['D', 'A', 'F', 'C', 'B', 'E']
GraphConnected	Degree Heuristic	0.159	1.000	['D', 'F', 'C', 'A', 'E', 'B']
GraphBipartite	Brute Force	3.326	1.0	['D', 'H', 'E', 'F', 'G']
GraphBipartite	Approximation	0.083	1.600	['D', 'H', 'G', 'A', 'F', 'B', 'I', 'E']
GraphBipartite	Degree Heuristic	0.058	1.200	['D', 'H', 'C', 'E', 'F', 'A']
GraphBig	Brute Force	35294.014	1.0	['F', 'T', 'D', 'K', 'H', 'B', 'E', 'R', 'P', 'N', 'L', 'J', 'Q']
GraphBig	Approximation	0.158	1.538	['D', 'A', 'K', 'J', 'H', 'G', 'C', 'B', 'E', 'F', 'T', 'S', 'R', 'Q', 'P', 'M', 'N', 'O', 'I', 'L']
GraphBig	Degree Heuristic	0.741	1.154	['K', 'H', 'E', 'N', 'Q', 'D', 'T', 'B', 'R', 'P', 'O', 'F', 'G', 'I', 'J']

Table 6: Trial 6

Execution

In order to generate the tables seen in the results section, the included script was run 6 times and the generated table saved each time. In order to add more graphs for testing, a new dictionary must be added to the script in which the keys are strings representing the vertex names and the values for a given key are the vertices which that key vertex is connected to. With this graph defined, all that is left is to pass it to one of the functions whose name starts with "vertex_cover_" in order to see get a cover for the graph. To include the graph in the results tables generated above, a new variable must be created that represents a row within the table. This new variable must include two hardcoded entries. The first being the name of the graph and the second indicating the algorithm used to find the cover. Then the algorithm must be run using the timeit.timeit() method in order to get the execution time. This time is then added to the row data. If an approximation algorithm is being used, that the approximation ratio must be calculated using the length of the cover returned divided by the length of the optimal cover. This approximation ratio is then added to the row data. Finally, the cover found by the algorithm is added to the row data. The row data is then added to the table. The order of rows in the table is determined by the order in which row data is added to the table. There is also a helper function that was implemented called plot_graph(graph, path) that takes a graph and creates an image of this graph. The images created are not always the drawing of a graph as they do not respect any special characteristics, like a graph being bipartite, but they are useful in making sure the graph has been defined properly.

Charles Sedgwick

Algorithm Analysis

I. Brute Force Algorithm

Brute force algorithm relies on several helper functions in the process of finding the minimal cover. The first is generate_edges(graph) which takes a graph, defined as a dictionary as described above, and returns a set of (vertex, neighbour) tuples used to represent and edge in the graph. This function uses two nested for loops to build this set. The outer loop iterates over the nodes in graph and the inner loop iterates over the set of neighbours to which the current node is connected to. The inner loop adds a edge to the set of edges if there is no edge in the set that described the undirected edge between the current node and the current neighbour. As implemented, the generate_edges(graph) function has a running time of $O(|graph.V|^2)$ since the worst case is a graph in which each node is connected to every other node requiring the outer and inner loops to iterate $|graph.V|^2$ times. Another helper function used by the brute force algorithm is verify_vertex_cover(cover,graph) which verifies that cover is a vertex cover for the provided graph. It also uses two nested for loops to determine if a set of vertices is indeed a cover for the provided graph. The outer loop iterates over the edges in the graph while the inner for loop iterates over the vertices in the suspected cover. If an edge is found that has neither of its vertices present in the cover than the function return false indicating that the set of vertices provided is not a cover. If at least one vertex in every edge is found in the suspected cover then the function returns true indicating that the provided cover is valid. This function runs in O(nm) time where n is the number of edges in the graph and m is the number of vertices in the cover. This running time is the result of two nested for loops: the outer loop iterates over the edges and the inner loop iterates over the vertices in the cover.

The last helper function that the brute force algorithm relies on is gen_subset(set, i) which returns all subsets of the set of vertices given that are of size i. The gen_subsets(set, i) function is a wrapper function that sets up the required variables then calls a recursive function call generate_subsets(set_, curr_subset, subsets_, k, next_index). The base case for this recursive function is when the size of curr_subset is equal to k. If the size of curr_subset is not k and next_index + 1 is less than or equal to the length of set_ then a copy of curr_subset is made called curr_subset_exclude, the element in set_ at next_index is added to curr_subset, and two recursive calls are made. The first

recursive call passes curr_subset as the current subset being built and the second call passes curr_subset_exclude as the current subset. This has the effect of creating branches in the recursion tree for both the subset that includes the element in set_ at next_index and subset that excludes that element. In this way, the function uses the divide and conquer approach to generate the subset by breaking the problem into the smaller problem of creating subsets of size less than k. The recurrence that describes this function is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = k, \\ 2T(n+1) + D(n) & n < k. \end{cases}$$

where D(n) is the time required to create the subsets passed to the recursive calls.

Since the recursion tree for the recurrence has k levels all with cost $2^i(c(n+i))$ resulting in total cost of for an input size of n of $T(n) = (n)(2^{(n-1)})(c((n-1)+(n-1))) + 2^n(c(T(1))$ which reduces to $T(n) = 2^n n^2 c - 2^n nc + 2^n c$. The $2^n n^2 c$ term dominates and in fact the 2^n component dominates resulting in a $O(2^n)$ upper bound for the function gen_subsets(set, i).

The effect of this exponential time in generating subsets is apparent in the results for the brute force algorithm as the size of the graph grows beyond handful of vertices. As the results show though, there are approximation algorithms that perform much better while still giving a cover that is at most 2 times the size of the optimal cover. Cormen et al. present the approximation algorithm APPROX-VERTEX-COVER and show that it is a polynomial time 2-approximation algorithm (Cormen, 2009). It is used here to compare the results of the vertex_cover_degrees(graph, cover) approximation algorithm.

The approximation algorithm vertex_cover_degrees(graph, cover) appears to run some polynomial time constant slower than vertex_cover_approx(graph) but the covers it returns seem to be as big or smaller than those that vertex_cover_approx(graph) returns. The vertex_cover_degrees algorithm adds only at most 1 vertex to the cover being built each iteration whereas the vertex_cover_approx algorithm adds at most 2 vertices. Algorithm vertex_cover_degrees iterates over the vertices in graph whereas algorithm vertex_cover_approx iterates over the edges of a graph. Since only one vertex is added to the cover each iteration for vertex_cover_degrees where as two are added to the cover in vertex_cover_approx and they share the same lower, the loose upper bound for the size of the vertex cover returned by vertex_cover_degrees is 2.

Discussion

In implementing the algorithms used in this work, I learned much more about Python since it is not a language I have used much at all. I learned how to manipulate data structures like dictionaries, how arguments are passed to functions and libraries for timing as well as data plotting. Coming from a C background, the data structures and the methods used to pass them to functions required the most effort to understand but, thanks to the vast amount of documentation that exists for Python, this did not end up being a time consuming issue. From theoretical standpoint, I became more comfortable in reasoning about graphs, determining the running time of algorithms that use both recursion and loops, and determining the approximation ratio for an approximation algorithm. These topics required considerable effort to apply to the problem given my relatively shallow background in math and graph theory.

Works Cited

Cormen, T. H., Leiserson, C. E., Rivest, R.L., and Stein, C. 2009. *Introduction to Algorithms, Third Edition* (3rd. ed.). The MIT Press.