Rüzgar Ayan 21801984

2020 FALL CS473 Programming Assignment 1 Report

Implementation Details

While parsing the graphs from the ".mtx" files, I have used my own Linked List implementation for the adjacency lists of the vertices. After reading all the edges from the file, I have compacted these adjacency lists into two one-dimensinoal arrays "int* vertices" and "int* edges" as mentioned in the slides.

Other than the Linked List, I have also implemented a MaxHeap data structure that supports building a heap from an array, extracting the max and decreasing or increasing a key for the 1.a) part of the assignment.

In the beginning of the KL algorithms, I have chosen to partition the vertices as the first half and the second half. I didn't want to partition it randomly since it doesn't even have many advantages but it makes debugging much harder. After this, my implementation of KL directly follows the one given in the pseudo-code and there is not much to explain. The biggest problem I had was to keep track of the vertices in the heaps since building a heap was changing their order in the array. To solve this, I've first initialized an array that holds all the vertices, their data and their hIndex. Then I have created two separate array for the sets A and B by pointing to the elements of the original array. This way, even though I built heaps from A and B, I could reach the information of any vertex from the original array.

Asymptotic Running Times

Since the number of passes are not certain, I will give the running times of a single pass. Also, assume that the maximum degree of the input graph is limited to a constant C as asked.

For 1.a), we first compute D values for all vertices which is O(|V|*C) or O(|E|). Then we build two heaps from the partition, each for O(|V|/2). Then in the inner loop that runs (|V|/2) times, we do two extractMax that costs O(log|V|) and at most 2*C increaseKey or decreaseKey operations which each cost O(log|V|). The rest is just finding the best number of swaps and doing the swaps that is simply O(|V|). In total, the runtime per pass is O(|E|+|V|*log|V|*C). Also, we calculate the initial and final cutsizes for the output and these take O(|E|) each.

For 1.b), we do not need to build heaps and also changin the D values take O(1) time compared to O(log|V|) in part 1.a). However, finding the vertices a and b with the maximum D values require us to go over all the vertices which takes O(|V|) time in a loop that runs O(|V|/2) times. This is the differentiating point between the two parts and the runtime per pass becomes O(|E|+|V|*|V|), quadratic in |V|. Or we can directly write O(|V|*|V|) as |V|*|V| dominates |E|.

Comparison with NetworkX

INPUT GRAPH	INITIAL CUTSIZE WITH 1.A)	FINAL CUTSIZE WITH 1.A)	RUNTIME WITH 1.A)	FINAL CUTSIZE WITH NETWORKX	RUNTIME WITH NETWORKX
Erdos02.mtx	**4582**	**1280**	**0.026 s**	**1144**	**0.213 s**
com-DBLP.mtx	**310172**	**100826**	**11.419 s**	**122607**	**96.047 s**
rgg_n_2_20_s0.mtx	**8911**	**4346**	**17.648 s**	**337669**	**600.671 s**

In all cases, my algorithm runs faster than the python implementation. But this is expected since we had a simplified version. In the first input graph, my algorithm finds a larger final cutsize than the python, this is again because we are using a simplified version and it cannot find the best final partition. The reason we have a better final cutsize in the second input graph is that python implementation doesn't go all the way, it has a max_iter parameter which is 10 by default. In the last input, surprisingly we have a very low initial cutsize and accordingly a low final cutsize. It turns out that this is because of the structure of this particular graph and the way I make the initial partition. Making the initial partition as the first half of vertices and the second half of vertices was already a good partition for the last graph.

Profiling and Results

After profiling 1.a), I have observed that about 80% of the time were used on the heap operations such as increaseKey, decreaseKey and heapify (for building the heaps). This was expected and there is not much to do to improve these and so I haven't do any improvements for 1.b) after profiling.

After profiling 1.b), I have observed that for an example execution of 8.83 seconds, 8.73 seconds of this was used while trying to find maximizing a and b from the two lists. This is not surprising as this operation was already the one defining the asymptotic running time of the whole algorithm. But this profiling showed me that if I will do any improvements, I must do these for finding the maximizing a and b faster.

Improvements

After observing that the maximum D values of unlocked vertices increase by at most 2 in every update, I have changed the code so that it can now find the maximizing a or b during the the updating process. When I increase the D value of some vertex by 2, I check if it is at least equal to (last maximum + 2), and if so I do not need to go over all the vertices. This gave me about 10 to 20 percent lower execution times depending on the size of the input. I did some other related improvements but they did not give that much performance, so I reversed these improvements to make the code more clear.