Munib Emre Sevilgen
21602416

CS 473 - Assignment 1
09.12.2020

1. Brief Explanation

The data kept for the algorithms:

- Graph by adjacency list implementation using the array with size V of linked lists.
- The arrays of vertex indexes with size V/2 for two partition A and B
- locations array with size V to keep the locations of the vertices at the array A or B with the information that the vertex is whether in A or B part.
- isLocked array with size V to keep that the vertices are locked or not.
- gain_A and gain_B arrays with size V/2 to keep the gain values. In the heap implementation, this array also keeps the index values of the vertices at A or B partition array.
- heap_indexes_A and heap_indexes_A with size V/2 to keep the heap indexes of the vertices at A and B partition.

In the imp_kl.c file, the first part of the code is for the adjacency list implementation of the graph. Then, there is the heap implementation with required functions and struct. There is the calculate_cut_value function to calculate the cut size of the partitioning. The readMatrixFile function is for reading the file and creating the graph by the data. It reads files by using an external library including mmio.c and mmio.h files. Then there are two different implementations of the simpler KL algorithm as simpler_KL and simpler_KL_heap. The first implementation is for the part 1.b) and the other one is for part 1.c). Then there is the main function to call the implementation files and print the appropriate output.

2. Discussion of 1.a) and 1.b)

a) For the sparse graphs, the worst-case running time of the algorithm is $\Theta(nlogn)$. Because updating gain values is $\Theta(logn)$ because of the heap implementation. Also, finding the maximum gain is $\Theta(logn)$.
For the dense graphs, the worst-case running time of the algorithm is $\Theta(n^2logn)$. Because if every vertex is adjacent to all vertices, then we should update the gain values for all vertex each time we found a maximum gain value. Updating gain values is $\Theta(nlogn)$ because of the heap implementation.

b) For the sparse graphs, the worst-case running time of the algorithm is $\Theta(n^2)$. Because finding the maximum gain is $\Theta(n)$.
For the dense graphs, the worst-case running time of the algorithm is $\Theta(n^2)$. Because if every vertex is adjacent to all vertices, then we should update the gain values for all vertex each time we found a maximum gain value. Updating gain values is $\Theta(n)$.

Therefore for the sparse graphs heap implementation is faster than the other. However, for the dense graphs, simple search implementation is faster than the heap implementation in the worst case.

3. Comparison of the Implementations

Running times (s)	Vertex counts	Heap implementation	Simple search implementation	NetworkX library implementation
Erdos02	6927	0.015625	0.296875	0.309001
com-DBLP	317080	5.734375	2453.046875	138.430240
rg_n_2_20_s0	1048576	11.640625	17439.718750	806.287238

By the running times, it can be said that the heap implementation is $O(nlogn)$ and the simple search algorithm is $O(n^2)$ as expected. The NetworkX implementation is also $O(nlogn)$ because it also uses the heap implementation. However, because of Python, it is slower than the C implementation.

Cut size	Initial cut sizes of my implementations	Heap implementation final cut size	Simple search implementation final cut size	NetworkX library implementation final cut size
Erdos02	4582	1206	1033	1082
com-DBLP	310172	115323	116120	112794
rg_n_2_20_s0	8911	4980	4997	335003

The heap and simple search implementations are very close to each other for all of the graphs. The NetworkX is also very similar to others for Erdos02 and com-DBLP graphs. However, for the rg_n_2_20_s0, the cut size of NetworkX is very large. The initial split algorithm can cause this problem. By default, the NetworkX goes the 10 iterations maximum and can't decrease the cut size enough.

4. Conclusions and Improvements

- I have used the locations array with size V as a hash-map to keep the locations of the vertices in A or B arrays and whether they are in the A and B partition. This reduces the searches for getting the information that the vertices are in A or B.
- In the profiler output, I see that the heap functions get_left_child, get_right_child, swap functions are called mostly. That is expected because of the heap structure. I have tried to simplify these functions in order to reduce the runtime.