ELTE Eötvös Loránd University

Faculty of Science

Searching and generating sparse (sub)graphs

Math project I. report

Bence Deák

Applied mathematics MSc

Supervisor:

Péter Madarasi

ELTE Institute of Mathematics

Department of Operations Research

Budapest
2024

The basics of (k, l)-sparsity

Let $G = (V, E)$ be a graph, and $k, l \in \mathbb{N}$ natural numbers.

Definition. *G is said to be (k, l)-sparse if any $X \subseteq V$ induces at most $\max\{k|X| - l, 0\}$ edges.*

Definition. *G is said to be (k, l)-tight if it is (k, l)-sparse and $|E| = k|V| - l$.*

The notion of (k, l)-sparsity and (k, l)-tightness generalizes the defining property of many interesting graph families. For example, it is well known that the edge set of a graph can be covered by k edge-disjoint forests if and only if the graph is (k, k)-sparse [1]. Another famous theorem in rigidity theory states the equivalence between $(2, 3)$-tightness and minimal rigidity [2]. Fortunately, there exist polynomial-time algorithms for finding a maximum size (or maximum weight) (k, l)-sparse subgraph of any given graph.

The pebble game algorithm

The pebble game algorithm for (k, l)-sparsity is perhaps the most widely used method for finding a spanning (k, l)-sparse subgraph with the maximum number of edges [3, 4, 5, 6]. Specifically, it can determine whether the whole graph is (k, l)-sparse. It can also be configured to handle the weighted case, where our goal is to maximize the total weight of the selected edges.

The algorithm starts with an empty graph, then iterates over the edges in an arbitrary order, and inserts each edge if and only if the resulting graph remains sparse. To check this condition efficiently, we have to maintain an orientation of the already inserted edges, then for each processed edge, we may need to reverse some (at most $l + 1$) paths in this digraph. This leads to an algorithm with a time complexity of $O(nm)$ (where n and m are the number of vertices and edges, respectively).

An improved version of the algorithm also maintains the components (i.e. the inclusion-wise maximal tight induced subgraphs). This way, to determine whether the current edge can be inserted, we only have to check if there is a component that contains both of its end vertices. Using the fact that two maximal (k, l)-tight induced subgraphs intersect in at most one vertex, we can achieve an $O(n^2)$ running time with a well-chosen data structure for the components [7].

Project work

Faster sparse graph generation

This project is closely related to another one, which focuses on building a modular and extensible library for generating all non-isomorphic graphs with certain properties (up to a given size). Accordingly, one of our main goals during the semester was to improve the filtering step of the algorithm for (k, l)-sparse graph generation. This filtering subroutine receives a graph $G = (V, E)$ that satisfies the given property, a new vertex $s \notin V$ to be inserted and its desired set of neighbours $N(s) \subseteq V$. The job of this function is to determine whether the graph $G' = (V + s, E \cup \{sv : v \in N(s)\})$ also satisfies the property. The generating procedure calls this subroutine for each possible $N(s)$, allowing global information to be shared between different function calls. A natural approach would be to naively check for each $X \subseteq N(s)$ whether there is a set $Y \subseteq V$ such that $X \subseteq Y$ and $i(Y) > k(|Y| + 1) - l - |X|$ (where $i(Y)$ is the number of edges induced by Y). One can observe that this algorithm is quite wasteful in the sense that for a given $X \subseteq V$, it goes through all $Y \supseteq X$ each time $N(s)$ is a superset of X. In fact, we could avoid iterating over all the supersets if we managed to efficiently calculate and store the "tightest" superset for every X (that is, a $Y \supseteq X$ that minimizes $k|Y| - i(Y)$). We can apply the following optimizations. Initially, we calculate the slack values $s(X) := k|X| - i(X)$, then compute all $s^*(X) := \min\{s(Y) : X \subseteq Y\}$ using (top-down) dynamic programming. This can be done in $O(n \cdot 2^n)$ time, where $n = |V|$. If the calls to the filtering subroutine are made in lexicographical order with respect to the binary representation of $N(s)$, then a similar (bottom-up) dynamic programming approach can be used to answer all calls in $O(n \cdot 2^n)$ time in total. This is a significant improvement over the $O(4^n)$ solution that naively tests the sparsity for all subsets of all possible neighbourhoods. The described algorithm currently exists only in theory, but I plan to write an efficient implementation of it during the next semester.

An efficient data structure for a pebble game heuristic

One of our aims during the semester was to come up with an asymptotically efficient data structure (which would be used in a heuristic for speeding up the pebble game algorithm) that supports the following operations on a graph $G = (V, E)$ with a weight function $w : V \to \mathbb{N}$ on the vertices (initially, $w \equiv 0$):

- Given a vertex $v \in V$, increase/decrease $w(v)$ by 1.
- Find an edge $uv \in E$ that maximizes $w(u) + w(v)$.

Let $n := |V|$ and $m := |E|$. A possible approach is to maintain a priority queue of the edges, where the priorities are the sum of the weights of the end vertices. Using the fact that the updates only change the priorities by 1, we can quickly arrive at a solution that performs each update in $O(\Delta(G))$ time and answers each query in amortized $O(1)$ time. This can be improved by applying a well-known technique known as square root decomposition. First, we divide V into "small" and "big" vertices based on their degrees: if $d(v) < \sqrt{m}$, then v is classified as small; otherwise, it is classified as big. We maintain a separate priority queue for the edges incident to each big vertex, and a priority queue for the rest of the edges. This way, both updates and queries can be performed in (amortized) $O(\sqrt{m})$ time. This is indeed an asymptotic improvement over the previous approach if $m = o(n^2)$. It remains unclear whether a more efficient solution exists; in the coming semesters, I plan to further investigate and attempt to optimize the complexity of this data structure.

Finding a maximum weight (k, l)-sparse subgraph in $O(n^2)$ time

Union pair-find is a data structure proposed by A. Lee, I. Streinu and L. Theran for efficiently maintaining the components in the component-based version of the pebble game algorithm. For a given input graph $G = (V, E)$, union pair-find has an $O(n^2)$ alleged running time (where $n = |V|$). Without such a data structure, the worst-case time complexity of the component-based pebble game algorithm would exceed the $O(n^2)$ bound stated in the introduction. The core of union pair-find is an $n \times n$ bit-matrix (one bit for each pair of vertices), the entries of which indicate whether there is a component that contains both vertices. Upon inserting an edge e, it may happen that a new component C arises (that contains e), and we need to merge some "old" components $C_1, \ldots, C_t$. This means, for each $u, v \in V(C)$, we have to set the corresponding entry in the bit-matrix to 1 (preferably without unnecessarily overwriting already set bits). Unfortunately, the complexity analysis of union pair-find in the original paper [7], as noted in [8, Page 339] and [9, Page 101-102], has a serious flaw: it falsely assumes that $|V(C_i) \cap (V(C_1) \cup \cdots \cup V(C_{i-1}))| \leq 1$ holds in each intermediate state while merging these components. At the beginning of this project, it was unclear whether the issue was solely with the analysis or with the data structure itself. Initially, I came up with an $O(n^{7/3})$ substitute solution, similar to union-pair find, which works by decomposing V into "small" and "big" vertices based on the number of components containing the given vertex. This approach is similar to square root decomposition, but the threshold is chosen to be $\Theta(n^{1/3})$. After some unsuccessful attempts at finding counterexamples for the alleged complexity of union pair-find, I conjectured that the flaw was in the analysis, while the data structure itself can be made truly efficient. As it turned out, it is possible to prove that the number of modifications in the bit-matrix throughout the entire execution of the algorithm is indeed $O(n^2)$. As each bit changes its value at most once, it is enough to give a bound for the number of unnecessary modifications (overwriting an already set bit); let us denote it by β. First, we need to observe that the sum $\sum_{i=1}^{t} |V(C_i) \cap (V(C_1) \cup \cdots \cup V(C_{i-1}))|$ is not too large: it can be bounded by $2t$. Using the fact that two different components intersect in at most one vertex, we can prove that

$$\beta = \sum_{i=1}^{t} \binom{|V(C_i) \cap (V(C_1) \cup \cdots \cup V(C_{i-1}))|}{2} \leq 4t^2.$$

As we merge only $O(n)$ components during the execution of the algorithm, the number of unnecessary modifications (and thus the number of modifications) sums up to $O(n^2)$ in total. The total running time of the algorithm is therefore $O(n^2)$. It is still a question whether some modification of this data structure can be used for improving the running time of generalized versions of this problem. I plan to investigate this further in the next semester.

References

[1] C. St. J.A. Nash-Williams. Edge-disjoint spanning trees of finite graphs. *Journal of the London Mathematical Society*, 1(1):445–450, 1961.

[2] G. Laman. On graphs and rigidity of plane skeletal structures. *Journal of Engineering mathematics*, 4(4):331–340, 1970.

[3] A. R. Berg. *Rigidity of Frameworks and Connectivity of Graphs.* PhD thesis, Aarhus University, Denmark, 2004.

[4] A. R. Berg and T. Jordán. Algorithms for graph rigidity and scene analysis. In G. Di Battista and U. Zwick, editors, *Algorithms-ESA 2003: 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003. Proceedings 11*, pages 78–89. Springer LNCS 2832, 2003.

[5] D. J. Jacobs and B. Hendrickson. An algorithm for two-dimensional rigidity percolation: the pebble game. *Journal of Computational Physics*, 137(2):346–365, 1997.

[6] A. Lee and I. Streinu. Pebble game algorithms and sparse graphs. *Discrete Mathematics*, 308(8):1425–1437, 2008.

[7] A. Lee, I. Streinu, and L. Theran. Finding and maintaining rigid components. *Canadian Conference on Computational Geometry*, 2005.

[8] P. Madarasi and L. Matúz. Pebble game algorithms and their implementations. In *12th Japanese-Hungarian Symposium on Discrete Mathematics and Its Applications March 21-24, 2023 in Budapest, Hungary*, 2023.

[9] A. Mihálykó. *Augmentation problems in count matroids and globally rigid graphs.* PhD thesis, Eötvös Loránd University, Budapest, 2022.