ELTE Eötvös Loránd University

Faculty of Science

Searching and generating sparse (sub)graphs

Math project II. report

Bence Deák

Applied mathematics MSc

Supervisor:

Péter Madarasi

ELTE Institute of Mathematics

Department of Operations Research

Budapest
2025

1 The basics of (k, l)-sparsity

For simplicity, we assume throughout this report that all graphs are loopless. Let $G = (V, E)$ be a graph, and let $k, l \in \mathbb{N}$ be natural numbers that satisfy $l < 2k$.

Definition 1. *G is said to be (k, l)-sparse if any $X \subseteq V$ induces at most $\max\{k|X| - l, 0\}$ edges.*

Definition 2. *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 most widely used such algorithm, praised for its simplicity and efficiency, is called the pebble game [3].

2 The naive pebble game algorithm

The pebble game algorithm relies on the following properties of (k, l)-sparsity.

Theorem 1 (Lorea [4])**.** *For a graph $G = (V, E)$ and $\mathcal{I} = \{F \subseteq E : (V, F)$ is (k, l)-sparse$\}$, $M = (E, \mathcal{I})$ is a matroid.*

Lemma 1 (Hakimi [5])**.** *Let $H = (V, F)$ be a (k, l)-sparse graph, with $u, v \in V$ two distinct vertices, and D a k-indegree-bounded orientation of H. Then:*

1. *If $\varrho_D(u) + \varrho_D(v) < 2k - l$, then $H' = (V, F \cup \{uv\})$ is (k, l)-sparse.*
2. *If $\varrho_D(u) + \varrho_D(v) \geq 2k - l$, and there is no path from $\{w : V \setminus \{u, v\} : \varrho_D(w) < k\}$ to $\{u, v\}$, then H' is not (k, l)-sparse.*

Remark 1. *Given a k-indegree-bounded orientation D of H, Lemma 1 also provides an algorithm for checking the sparsity of H'. While $\varrho_D(u) + \varrho_D(v) \geq 2k - l$, we find a path P from $\{w \in V \setminus \{u, v\} : \varrho_D(w) < k\}$ to either $\{u, v\}$. If no such path exists, then H' is not sparse. Otherwise, we reverse the arcs of P, which decreases $\varrho_D(u) + \varrho_D(v)$ while preserving the k-indegree bound of the orientation. By repeating this process, the sparsity of H' can be determined with at most $l + 1$ path reversals.*

Consider the more general problem of finding a maximum-weight sparse subgraph of a graph $G = (V, E)$. Assume the edges $e_1, \ldots, e_m$ are sorted in non-increasing order by weight. We construct an optimal subgraph $H = (V, F)$ iteratively: for each edge e_i, we add it to F if and only if the resulting graph $H' = (V, F \cup \{e_i\})$ remains sparse. Theorem 1 ensures that this greedy algorithm yields an optimal solution. To efficiently determine whether an edge can be added, we maintain a k-indegree-bounded orientation D, and for each edge, apply Lemma 1 along with Remark 1. If H' is sparse, we add e_i to D with an appropriate orientation to preserve the indegree bound. Since each of the m edges can be processed in $O(n)$ time, the total time complexity of the algorithm is $O(nm)$, where $n = |V|$.

3 The component-based pebble game algorithm

Suppose that we can check whether an edge can be added to H in $O(1)$ time using an oracle. Then, the naive pebble game would handle the rejected edges in $O(1)$ time and the accepted edges in $O(n)$ time, resulting in an $O(n^2 + m)$ time algorithm. In the following, we will show how to create such an oracle.

Definition 3. *A* (k, l)-block *of a (k, l)-sparse graph is a subset $X \subseteq V$ of the vertices that induces a (k, l)-tight subgraph.*

Definition 4. *A* (k, l)-component *of a (k, l)-sparse graph is an inclusion-wise maximal (k, l)-block.*

Lemma 2. *Let X and Y be blocks of a sparse graph $H = (V, F)$ such that $|X \cap Y| \geq 2$. Then $X \cap Y$ and $X \cup Y$ are also blocks.*

Corollary 1. *Two different components of a sparse graph can intersect in at most 1 vertex.*

Corollary 2. *By adding an edge uv to a sparse graph, at most one new component emerges.*

Corollary 3. *The total size of the components of a sparse graph H is $O(n)$.*

For a sparse graph H, the graph $H' = H \cup \{uv\}$ remains sparse if and only if there is no component $X \subseteq V$ that contains both u and v. Therefore, in order to decide whether the edge uv can be added, the algorithm must maintain the components of H throughout its execution. When an edge is inserted, it may create at most one new component and potentially require the removal of some existing ones. Without going into the implementation details, we arrive at the following algorithm.

Algorithm 1 Component Pebble Game

Input: An undirected graph $G = (V, E)$, with edges $e_1, \ldots, e_m$ sorted in non-increasing order of weight; two integers k and l such that $0 \leq l < 2k$.
Output: The edge set of an optimal (k, l)-sparse subgraph.

```
1:  procedure ComponentPebbleGame(G = (V, E), k, l)
2:      F ← ∅                                          ▷ Initialize the set of accepted edges
3:      D ← (V, ∅)                                     ▷ Initialize a directed graph on V without arcs
4:      for e ← e_1, ..., e_m do                       ▷ Iterate over the edges sorted from best to worst
5:          u, v ← endpoints(e)
6:          if not InCommonComponent(u, v) then        ▷ No component contains both u and v?
7:              while ϱ_D(u) + ϱ_D(v) ≥ 2k − l do
8:                  find a path P in D from {w ∈ V \ {u, v} : ϱ_D(w) < k} to {u, v}
9:                  reverse the arcs of P in D         ▷ Reduce the indegree of u or v
10:             end while
11:             F ← F ∪ {e}                            ▷ Accept edge e
12:             if ϱ_D(v) < k then
13:                 D ← D ∪ {uv}                       ▷ Add an arc from u to v in D
14:             else
15:                 D ← D ∪ {vu}                       ▷ Add an arc from v to u in D
16:             end if
17:             C ← FindComponent(D, u, v)             ▷ Find the new component if there is one
18:             if C ≠ ∅ then                          ▷ New component found?
19:                 UpdateComponents(C)                ▷ Update component data
20:             end if
21:         end if
22:     end for
23:     return F                                       ▷ Return the set of accepted edges
24: end procedure
```

Note 1. *The algorithm relies on the subroutines* InCommonComponent*,* FindComponent*, and* UpdateComponents*. The* InCommonComponent *procedure checks whether there exists a component X that contains both of the given vertices. The* FindComponent *subroutine returns the new component that emerges after adding the edge uv to H, or $\emptyset$ if no new component is formed. Finally,* UpdateComponents *is responsible for updating the data structures used to maintain the current set of components.*

In the following, we will focus on creating efficient implementations for the above subroutines. To do so, we need a few observations.

Lemma 3. *During the algorithm, at most $O(n)$ components emerge.*

Lemma 4. *After inserting an edge uv, a subset $X \subseteq V$ of vertices containing both u and v is a block if and only if $\varrho_D(X) = 0$, and $\varrho_D(w) = k$ for each $w \in X \setminus \{u, v\}$.*

Lemma 5. *After inserting the edge uv, let $S := \{w \in V \setminus \{u, v\} : \varrho_D(w) < k\}$, and define T as the set of vertices not reachable from S. A new component forms if and only if both u and v belong to T.*

Lemma 6. *Let T be as defined in Lemma 5. If the insertion of the edge uv results in a new component, then this component is precisely T.*

Using Lemmas 5 and 6, we can efficiently implement the FindComponent subroutine.

Algorithm 2

Input: A directed graph $D = (V, A)$ and two nodes $u, v \in V$.
Output: The component containing both u and v if it exists; $\emptyset$ otherwise.

procedure FindComponent($D = (V, A), u, v$)
    $T \leftarrow$ set of vertices not reachable from $\{w \in V \setminus \{u, v\} : \varrho_D(w) < k\}$ in D
    if $u, v \in T$ **then** ▷ New component emerged?
        return T
    else
        return $\emptyset$ ▷ Indicate that no new component is found
    end if
end procedure

The most natural data structure for maintaining the components is a bit matrix $M \in \{0, 1\}^{V \times V}$, where $M_{u,v} = 1$ if and only if there exists a component in H containing both u and v. In addition, we maintain a list $\mathcal{C}$ of the components, each represented as a list of vertices. If $k \neq l$, we start with $M = 0$ and an empty list $\mathcal{C}$. Otherwise, $M = I$, and $\mathcal{C}$ initially consists of singleton components, each containing a single vertex.

Throughout the algorithm, we maintain M and $\mathcal{C}$ so that they satisfy the following invariants with respect to the current graph H:

- $M_{u,v} = 1$ if and only if there is a component $X \subseteq V$ containing both u and v.
- The entries in $\mathcal{C}$ are exactly the components of H.

We are ready to implement the subroutines InCommonComponent and UpdateComponents.

Algorithm 3

Input: Two nodes $u, v \in V$.
Output: `true` if there exists a component including both u and v; `false` otherwise.

procedure InCommonComponent(u, v)
    return $M_{u,v} = 1$
end procedure

Algorithm 4

Input: A new component C.
Effect: Updates M and $\mathcal{C}$ to satisfy the required invariants.

procedure UpdateComponents(C)
    $U \leftarrow \emptyset$ ▷ Initialize the union of the components contained in C
    $\mathcal{C}' \leftarrow \{C\}$ ▷ Initialize the new set of components
    for $X \in \mathcal{C}$ **do**
        if $X \subseteq C$ **then** ▷ Component contained in C?
            for $(u, v) \in (U \setminus X) \times (X \setminus U)$ **do** ▷ Mark vertex pairs between U and X
                $M_{u,v} \leftarrow M_{v,u} \leftarrow 1$
            end for
            $U \leftarrow U \cup X$ ▷ Merge X into U
        else
            $\mathcal{C}' \leftarrow \mathcal{C}' \cup \{X\}$ ▷ Append X to $\mathcal{C}'$
        end if
    end for
    for $(u, v) \in U \times (C \setminus U)$ **do** ▷ Handle pairs with exactly one vertex in U
        $M_{u,v} \leftarrow M_{v,u} \leftarrow 1$
    end for
    for $(u, v) \in (C \setminus U) \times (C \setminus U)$ **do** ▷ Handle pairs completely outside U
        $M_{u,v} \leftarrow 1$
    end for
    $\mathcal{C} \leftarrow \mathcal{C}'$ ▷ Update $\mathcal{C}$ to contain the components of the new graph
end procedure

Note 2. *We represent U and $\mathcal{C}'$ as lists. By maintaining the characteristic vectors I_C, I_X, and I_U for the sets C, X, and U as needed, all Cartesian products (lines 6, 14, and 17) can be computed efficiently. The condition in line 5 can be checked in constant time by verifying that the first two vertices of X belong to C.*

It is clear that, excluding the running time of the UpdateComponents subroutine, Algorithm 1 runs in $O(n^2 + m)$ time. We now turn to analyzing the time complexity of the UpdateComponents subroutine.

Observation 1. *Each time the condition at line 5 is satisfied, we have to construct the characteristic vectors and update U, which takes $O(n)$ time. By Lemma 3, this condition is met at most $O(n)$ times throughout the algorithm. According to Corollary 3, the updates in line 11 require $O(n)$ total time per subroutine call. Since the subroutine is called $O(n)$ times, the overall cost of these updates is $O(n^2)$.*

The only part that remains unclear is whether the total size of the Cartesian products, or equivalently, the total number of modifications in M, is also $O(n^2)$. To address this, we make a crucial observation:

Lemma 7. *Suppose a new component C is formed after inserting the edge uv, and let $C_1, \ldots, C_t$ be the components that are removed. Define $U_0 := \emptyset$, and for each $i = 1, \ldots, t$, let $U_i := U_{i-1} \cup C_i$. Then the following inequality holds:*

$$\sum_{i=1}^{t} 2 \cdot \binom{|U_{i-1} \cap C_i|}{2} \leq 4t^2$$

Proof. We know that $i(U_t) \leq k|U_t| - l$ and $i(C_i) = k|C_i| - l$ for each i. From $i(U_t) \geq \sum_{i=1}^{t} i(C_i)$, we get

$$k \sum_{i=1}^{t} |C_i| - tl = \sum_{i=1}^{t} i(C_i) \leq i(U_t) \leq k|U_t| - l$$

After rearranging the terms,

$$\sum_{i=1}^{t} |U_{i-1} \cap C_i| = \sum_{i=1}^{t} (|C_i| - |C_i \setminus U_{i-1}|) = \sum_{i=1}^{t} |C_i| - |U_t| \leq \frac{l}{k} \cdot (t-1) \leq 2(t-1).$$

By squaring both sides and applying $\sum {x_i}^2 \leq \left(\sum x_i\right)^2$ to the left-hand side, the inequality follows. □

Lemma 8. *The matrix M is modified $O(n^2)$ times throughout the algorithm.*

Proof. The total number of modifications is $\alpha + \beta$, where α denotes the number of changes from 0 to 1, and β denotes the number of changes from 1 to 1, which we refer to as *redundant*. Clearly, $\alpha = O(n^2)$.

Claim 1. *Let C be the new component formed during a call to* UpdateComponents*, and let $C_1, \ldots, C_t$ be the components that are removed. Define $U_0 := \emptyset$ and for each $i = 1, \ldots, t$, let $U_i := U_{i-1} \cup C_i$. Then the number β_C of redundant modifications made during this subroutine call is at most $n + \sum_{i=1}^{t} 2 \cdot \binom{|U_{i-1} \cap C_i|}{2}$.*

Proof. Redundant modifications are performed only by loops on line 6 and line 17. In the latter one, a redundant modification can only happen on the main diagonal.

We first consider the modifications on the main diagonal. These are made exclusively by the loop on line 17. As each diagonal entry is modified at most once, the total number of such modifications is at most n.

Next, consider the modifications outside the main diagonal. Let $u, v \in C$, with $u \neq v$, be distinct vertices such that $M_{u,v}$ is set to 1 by the subroutine, even though it was already 1 before the call. This implies that there was a former component X containing both u and v, so $|X \cap C| \geq 2$. By Lemma 2, it follows that $X \cup C$ is a block. Since C is maximal, we must have $X \subseteq C$, meaning $X = C_p$ for some p. The number of such pairs $(u, v) \in C_p$ corresponds precisely to the pth term of the above sum. Hence, the total number of modifications outside the main diagonal is $\sum_{i=1}^{t} 2 \cdot \binom{|U_{i-1} \cap C_i|}{2}$. □

Let $\mathcal{C}^*$ be the set of all components created during the algorithm. By Lemma 2, we have $\beta_C \leq 4t_C^2 + n$ for each $C \in \mathcal{C}^*$, and Lemma 3 implies that $\sum_{C \in \mathcal{C}^*} t_C \leq |\mathcal{C}^*| = O(n)$. Therefore, the total number of redundant modifications is $\beta = \sum_{C \in \mathcal{C}^*} \beta_C = O(n^2)$. □

We note that a similar component-based data structure, known as union pair-find, was introduced by Lee, Streinu, and Theran in [6]. However, its complexity analysis contained serious flaws, as pointed out in [7, Page 339] and [8, Pages 101–102].

4 The $2k \leq l < 3k$ case

Throughout this chapter, we consider only simple graphs. In the range $2k \leq l < 3k$, the sparsity condition is required only for vertex sets of size at least three. Under this definition, the family of edge sets of sparse subgraphs no longer forms a matroid, so the greedy algorithm is not guaranteed to yield an optimal solution. However, we can still construct an inclusion-wise maximal (k, l)-sparse subgraph using a similar approach, based on the following observation:

Lemma 9. *Let $H = (V, F)$ be a (k, l)-sparse graph, and let $u, v \in V$ be distinct vertices such that $uv \notin F$. Then the graph $H' := (V, F \cup \{uv\})$ is (k, l)-sparse if and only if, for every vertex $w \in V \setminus \{u, v\}$, there exists a k-indegree-bounded orientation D of H such that $\varrho_D(u) + \varrho_D(v) + \varrho_D(w) < 3k - l$.*

Using the above lemma, a straightforward modification of the naive pebble game algorithm yields an $O(n^2m)$ time solution. This can be improved to $O(n^3 + m)$ by efficiently maintaining the components of our sparse subgraph. Alternatively, we can make use the following lemma:

Lemma 10. *Let $H = (V, F)$ be a (k, l)-sparse graph, and let $u, v \in V$ be distinct vertices such that $uv \notin F$. Then the graph $H' := (V, F \cup \{uv\})$ is (k, l)-sparse if and only if the following conditions hold:*

- *There exists a g-indegree-bounded orientation $D = (V, A)$ of H, where $g(u) = g(v) = 0$ and $g(w) = k$ for all $w \in V \setminus \{u, v\}$.*
- *The digraph $D' = (V \setminus \{u, v\} \cup \{s\}, A')$ is t-arc-connected from the root s, where $t := l + 1 - 2k$, and A' is obtained from A by adding $k - \varrho_D(w)$ parallel arcs from s to each $w \in V \setminus \{u, v\}$.*

Proof.

Sufficiency: Let $w \in V \setminus \{u, v\}$ be an arbitrary vertex, and let $P_1, \ldots, P_t$ be arc-disjoint paths from s to w. By removing the first arc of each path, we obtain t arc-disjoint paths $P'_1, \ldots, P'_t$ in D, all ending at w, such that each vertex $x \in V \setminus \{u, v\}$ is the starting point of at most $k - \varrho_D(x)$ paths. Reversing the arcs of all P'_i paths yields an orientation that satisfies the conditions of Lemma 9.

Necessity: Assume that H' is sparse. For contradiction, suppose that H does not have a g-indegree-bounded orientation. Then, by the orientation lemma [5], there exists a subset $X \subseteq V$ such that $i_H(X) > g(X)$. If $|X| \geq 3$, then $i_H(X) > g(X) \geq k|X| - l$, contradicting the sparsity of H. On the other hand, if $X = \{u, v\}$, then $i_H(X) > 0$, which contradicts the assumption that $uv \notin F$.

We now show that D' is t-arc-connected from the root s. Let $D'' = (V \cup \{s\}, A'')$ be the digraph obtained from D by adding $k - \varrho_D(w)$ parallel arcs from s to each vertex $w \in V \setminus \{u, v\}$. Let $T' \subseteq V \setminus \{u, v\}$ be arbitrary, and define $T := T' \cup \{u, v\}$. By the sparsity of H', we have $i_{D''}(T) \leq k|T| - l - 1$. Therefore,

$$\varrho_{D'}(T') = \varrho_{D''}(T) = \sum_{w \in T} \varrho_{D''}(w) - i_{D''}(T) \geq k(|T| - 2) - (k|T| - l - 1) = t.$$

□

For a fixed $\kappa \in \mathbb{N}$, the rooted κ-arc-connectivity of a digraph with $O(n)$ arcs can be computed in $O(n)$ time when $\kappa \leq 2$ [9], and in $O(n \log n)$ time when $\kappa > 2$ [10]. Consequently, by the above lemma, the naive algorithm can be modified to solve the inclusion-wise maximal sparse subgraph problem in $O(nm)$ time when $l \leq 2k + 1$, and in $O(nm \log n)$ time when $l > 2k + 1$.

5 Summary

One of the main results of the semester is a detailed description of the algorithms for the maximum (k, l)-sparse subgraph problem, presented in a paper soon to be published. This includes the quadratic-time algorithm developed during the past semester. In terms of mathematical novelty, another result is the design of an efficient algorithm for finding an inclusion-wise maximal (k, l)-sparse subgraph in the case $2k \leq l < 3k$.

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

[4] M. Lorea. On matroidal families. *Discrete Mathematics*, 28(1):103–106, 1979.

[5] S. L. Hakimi. On the degrees of the vertices of a directed graph. *Journal of the Franklin Institute*, 279(4):290–308, 1965.

[6] A. Lee, I. Streinu, and L. Theran. Finding and maintaining rigid components. 2005.

[7] 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.

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

[9] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 246–251, 1983.

[10] H. N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 112–122, 1991.