Information and Computation 289 (2022) 104951

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/yinco

Linear-time parameterized algorithms with limited local resources ☆

Jianer Chen [a,b,*], Ying Guo [a], Qin Huang [b]

[a] School of Computer Science, Guangzhou University, Guangzhou 510006, PR China
[b] Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77843, USA

ARTICLE INFO

Article history:
Received 3 March 2020
Received in revised form 6 May 2022
Accepted 21 August 2022
Available online 28 August 2022

Keywords:
Bigdata
Linear-time algorithm
Space complexity
Graph matching

ABSTRACT

We propose a new computational model for the study of massive data processing. Our model measures the complexity of reading the input data in terms of their very large size N and analyzes the computational cost in terms of a parameter k that characterizes the computational power provided by limited local computing resources. We develop new algorithmic techniques for solving well-known computational problems on the model. In particular, randomized algorithms of running time $O(N+g_1(k))$ and space $O(k^2)$, with very high probability, are developed for the famous graph matching problem on unweighted and weighted graphs. More specifically, our algorithm for unweighted graphs finds a k-matching (i.e., a matching of k edges) in a general unweighted graph in time $O(N+k^{2.5})$, and our algorithm for weighted graphs finds a maximum weighted k-matching in a general weighted graph in time $O(N+k^3\log k)$.

1. Motivations

Recent progress in data science has shown that classical algorithmic techniques may become inadequate when dealing with data sets of enormous size. For example, Facebook has billions of users and trillions of links [21]. Thus, a traditionally "efficient" algorithm of running time say $O(n^2)$ may turn out to be not practically feasible. There have been fast growing interests in the study of massive data sets. The research has included the study of structures of massive data and data queries (e.g., [15]), parallel and distributed processing of massive data (e.g., [23]), and preprocessing of massive data (e.g., [14]). The research has been driven directly by practical applications in massive data processing, and is essentially heuristic-based. There has also been very active research in the algorithmic community. The study of very fast (sublinear-time, linear-time, or nearly linear-time) algorithms in dealing with massive data sets has drawn extensive attention. A number of computational models for dealing with massive data sets have been proposed and studied. In particular, data *streaming* and *semi-streaming* models [24,16] have been proposed and studied, where the massive data (e.g., "big graphs") may dynamically change and the algorithms must process the input stream in the order it arrives while using only a limited amount of memory. Very recently, studies on streaming algorithms based on the framework of parameterized computation have appeared [6,5,7].

In the current paper, we propose a new (theoretical) computational model for the study of massive data processing with limited "local" computing resources. Our model is of a multivariate nature, which measures the complexity of reading the

☆ This work is supported in part by the National Natural Science Foundation of China under grant 61872097.
* Corresponding author.
E-mail address: chen@cse.tamu.edu (J. Chen).

https://doi.org/10.1016/j.ic.2022.104951
0890-5401/

very large data sets in terms of the size of the data sets and analyzes the computational cost in terms of a parameter that characterizes the computational power provided by limited local computing resources. In particular, problems in our consideration have two parameters N and k, where N is the input size, which is assumed to be extremely large thus superlinear-time (such as quadratic-time) algorithms would be considered impractical, while k gives the "size" of feasibility such that the limited local computing resource (e.g., a normal computer) can handle problems with complexity (time and space) bounded by a polynomial of k, in addition to the linear-time reading from the input data. More specifically, we will study algorithms for processing massive data sets that run in *linear-time* in terms of the input size N, and polynomial time and polynomial space in terms of the parameter k, i.e., algorithms running in time $O(N + k^{O(1)})$ and space $O(k^{O(1)})$.

We argue that the proposed model is theoretically interesting and practically meaningful. Insisting on strict linear time (which is necessary to read the input) in terms of the size of input data sets allows us to process data sets of very large size. On the other hand, there seems to be no simple functional relations between the size of input data sets and the power of available computational resources. In many cases, problems in massive data processing (such as aggregations) on very large data sets are looking for solutions of size manageable by local computational resources, where the size of solutions and the size of input data sets are unnecessarily related proportionally. Therefore, it is meaningful and convenient by introducing another parameter k to characterize the available computational resources. The constraint on the space complexity in terms of the parameter k reflects the fact that although massive data sets are stored publicly, users can only read the data but do not own the space for storing the data. Allowing the cost of local resources to be bounded by polynomials of the parameter k offers new challenges in algorithmic research. We point out that optimizing the cost of local resources in terms of the parameter k implies widening the applicability of the algorithms. For example, if k is the solution size, then algorithms whose resources are bounded by lower-degree polynomials of k allow us to handle massive data problems with larger solutions.

As examples, we consider a number of well-known problems that have been extensively studied in algorithmic research, and demonstrate how these problems can be solved in the proposed model. In particular, we show how the famous graph matching problems (on general unweighted and weighted graphs) can be solved on this model. We present a randomized algorithm that finds a k-matching (i.e., a matching of k edges) in a general unweighted graph in time $O(N + k^{2.5})$ and a randomized algorithm that constructs a maximum weighted k-matching in a general weighted graph in time $O(N + k^3 \log k)$. Both algorithms have very high success probability and have their space complexity bounded by $O(k^2)$.

2. Definitions and related work

Let A be an algorithm that solves a computational problem Q. Inputs to the algorithm A take the form of pairs (x, k), where x is a proper encoding of an instance of Q and k is a parameter. For example, inputs to an algorithm that solves the *maximum weighted k-matching* problem are of the form (G, k), where G is a weighted graph given in an adjacency list, encoded properly, and the instance is looking for a k-matching in G that has the maximum weight over all k-matchings in the graph G.

We assume that our algorithms run on the word-RAM model, in which each basic operation (e.g., arithmetic operations and comparison) on words (i.e., the basic elements in a problem instance) takes constant time. Moreover, we assume that the instances are "word addressable" so that algorithms can read any word in an input instance in constant time. On the other hand, we do not allow algorithms to write (i.e., to modify) on input data. We will be focused on algorithms whose running time is bounded by $O(N + k^{O(1)})$ and whose space is bounded by $O(k^{O(1)})$, measured in word complexity, where N is the "size" of the input, i.e., the number of words in the input instance, and k is a parameter independent of the input size N that measures the "local complexity" of the algorithms. We remark that by definition, our algorithms will run in linear-time in the size of the input data for *both* word complexity and bit complexity. In fact, under common assumptions, the N words in the input can be given in $O(N \log N)$ bits. Since each basic word operation takes constant time in word complexity, which is $O(\log N)$-time in bit complexity, the $O(N)$-time word complexity of the algorithms implies $O(N \log N)$-time bit complexity, which is linear in terms of bit complexity of the input data. On the other hand, the word complexity $O(k^{O(1)})$ in local time and space would have an additional $\log N$ factor if we use bit complexity. We remark that, unlike some other proposed models (e.g. [6,24]), the complexity bounds given for an algorithm in our model are not allowed to have an "implicit" poly-log (i.e., a polynomial of $\log N$ or $\log k$) factor. Therefore, the time complexity of an algorithm in our model counts up the number of "normal" computational steps of the algorithm.

There have been several computational models in the literature that are related to our model.

A well-known complexity class SC (Steve's Class) that bounds both time and space complexities simultaneously was proposed by Stephen Cook [8], which consists of problems that are solvable in polynomial time while, *simultaneously*, with the space being bounded by $O(\log^{O(1)} n)$. In particular, the set of deterministic context-free languages is in the class SC [8]. Because the model allows high-degree polynomials in its running time, it may not be suitable for the paradigm of massive data processing.

Motivated by massive data processing, sublinear-time algorithms have been studied recently [29], which use randomization and inspect only a portion of the input data to give (in some sense imprecise) solutions. Quality of sublinear-time algorithms is measured in terms of the input size N and an error bound ϵ. On the other hand, linear time and quasi-linear time algorithms have been the focus in algorithmic research for years and have been studied extensively [9], where however a simultaneous bound on space complexity was seldom considered.

In the study of parameterized computation, there have been recent interests in "linear-time kernelization" algorithms [25]. A kernelization algorithm for a parameterized problem Q translates an instance (x, k) of Q into an "equivalent" instance (x', k') of Q such that both $|x'|$ and k' are bounded by a function of k. In particular, linear-time kernelization algorithms for maximum matching in unweighted graphs have been developed in terms of various parameters of the input graph, such as the feedback edge number, the feedback vertex number, and the distance to chain graphs [25]. However, the kernelization algorithms given in [25], as well as in other recent work in this direction, operate on the input graphs. As a result, the working space of the algorithms is a function of the size N of the input. There has also been recent research on parameterized and kernelization algorithms for NP-hard problems on dynamic inputs whose major concern is on bounding the update time by $f(k)N^{1+o(1)}$ for a function $f(k)$ of k [1].

The study of streaming/semi-streaming algorithms, in particular such algorithms for graph matching, has attracted much attention in recent years (see, e.g., [24,28,16,12] and their references). In such algorithms, input data are given as a stream of the data while the algorithms must process the input data in the order they arrive, within a given space bound. For instance, streaming/semi-streaming graph algorithms in general are restricted to space bound $O(n \log^{O(1)} n)$, where n is the number of vertices in the input graph (thus can be sublinear in terms of the size of the graph). Another complexity measure in streaming/semi-streaming algorithms is the "per-element process-time" (i.e., the *update time*) [28], which, when multiplied by the number of elements in the input, gives an upper bound on the running time of the algorithms.

More recently, there are increasing interests in the study of streaming algorithms under the framework of parameterized computation. Fafianie and Kratsch [13] considered polynomial-time kernelization algorithms for streaming graphs on a number of NP-hard problems, where the algorithms are restricted to have space bounded by $O(k^{O(1)})$. Classifications of parameterized streaming algorithms in terms of space complexity have been studied in [4]. Parameterized streaming algorithms for graph matching on the dynamic streaming model (in which the stream consists of both edge insertion and edge deletion operations) have been studied, where the focus is on the graph k-matching problem (i.e., constructing a k-matching in an unweighted graph or constructing a maximum weighted k-matching in a weighted graph) [5–7]. A lower bound on space complexity for streaming algorithms for the graph k-matching problem has been derived [7]. Under the condition that no graphs in the stream contain matchings of more than k edges, randomized streaming algorithms of $O(k^2 \log^{O(1)} N)$ space and $O(\log^{O(1)} N)$ update time for the graph k-matching problem have been developed for unweighted graphs [6,7]. Streaming algorithms for the k-matching problem on unweighted graphs without the above assumption on the graph stream, and on weighted graphs, have also been proposed and studied, but the space and update time complexities of the algorithms are significantly worsened [5–7]. Very recently, streaming algorithms on the dynamic streaming model for the k-matching problem on unweighted and weighted graphs with improved update time have been developed [3].

3. Case study I: matching in unweighted graphs

In this and the next sections, we provide thorough investigations on algorithms in our proposed model that solve the famous graph matching problems. This section is focused on unweighted graphs, while the next section is on weighted graphs.

All graphs in our discussion are undirected, which are given in the adjacency list format. A graph G is *weighted* if each edge in G is associated with a *weight*, which is a real number.

A *matching* M in a graph G is a set of edges in G such that no two edges in M share a common end. A matching is a *k-matching* if it consists of exactly k edges. A vertex v is *covered* by the matching M if v is an end of an edge in M. Otherwise, the vertex v is *uncovered*.

The instances of the (parameterized) Unweighted Graph Matching problem (p-UGM) are pairs of the format (G, k), where G is an unweighted graph and k is an integer (the parameter). An algorithm that solves the p-UGM problem on an input (G, k), either returns a k-matching in the graph G, or reports that no k-matching exists in G.

Throughout this paper, we will let $N = |V| + |E|$ be the "size" of a graph $G = (V, E)$.

We remark that the trivial greedy algorithm that finds a maximal matching, i.e., the algorithm that repeatedly adds edges with uncovered ends to the matching, cannot be directly used in our model: to check if an end of an edge is uncovered, we need to search in the vertices that are already covered, which will take time up to $O(\log k)$, resulting in an algorithm whose running time is at least $O(N \log k)$.

Let G be a graph and let k be an integer. A vertex v in G is a *large-vertex* if the degree of v is at least $2k$. A vertex is a *small-vertex* if it is not a large-vertex.[1]

Lemma 3.1. *If a graph G has at least k large-vertices, then G has a k-matching, which can be constructed in time $O(N + k^2 \log k)$ and space $O(k)$.*

Proof. Let v_1, v_2, ..., v_k be k large-vertices in G. We simply pick k edges of the form $[v_i, w_i]$, where for each $1 \le i \le k$, the vertex w_i is not in the vertex set $Q = \{v_1, v_2, \ldots, v_k\} \cup \{w_1, w_2, \ldots, w_{i-1}\}$. Note that this is always possible since the

[1] We remark that the idea of classifying vertices by their degrees, although in different settings, has appeared in previous work on streaming algorithms for graph matching [6,12]. For example, in the development of (unparameterized) streaming algorithms to estimate the maximum matching size [12], vertices in a graph are divided into "heavy vertices" and "light vertices."

large-vertex v_i has at least $2k$ neighbors while the number of vertices in the set $Q \setminus \{v_i\}$ is $(k-1)+(i-1) \leq 2k-2$. Such k edges $[v_i, w_i]$, $1 \leq i \leq k$, obviously make a k-matching in the graph G.

To implement this, we scan the graph G to identify the first k large-vertices $v_1, v_2, \ldots, v_k$ in G and store them in the set Q in space $O(k)$. The set Q is organized as a balance search tree that supports searching and insertion in logarithmic time per operation. We then re-scan the graph G, and for each i, $1 \leq i \leq k$, we work on the large-vertex v_i. Inductively, we have the set $Q = \{v_1, v_2, \ldots, v_k\} \cup \{w_1, w_2, \ldots, w_{i-1}\}$ stored in space $O(k)$. Since the set $Q \setminus \{v_i\}$ has no more than $2k-2$ vertices, to find an edge $[v_i, w_i]$ where w_i is not in the vertex set Q, we need to examine at most $2k-1$ neighbors of v_i. After finding the edge $[v_i, w_i]$, we add the vertex w_i to the set Q, thus completing the process on the i-th large-vertex v_i. As a result, finding the edge $[v_i, w_i]$ takes at most $O(k)$ searching/insertion operations on the set Q, which is done in time $O(k \log k)$. In conclusion, it takes time $O(N + k^2 \log k)$ and space $O(k)$ to construct the k-matching $\{[v_1, w_1], [v_2, w_2], \ldots, [v_k, w_k]\}$ in the graph G. □

Now we consider the situation where the graph G has only h large-vertices $v_1, v_2, \ldots, v_h$, where $h < k$. An *h-reduced graph* G_h of G is constructed from G, using the following procedure:

1. For each large-vertex v_i: pick arbitrary $\deg(v_i) - 2k$ edges of the form $[v_i, w_i]$, where w_i is a small-vertex, and delete these edges.
2. Delete all vertices of degree 0 in the resulting graph.

We give some remarks on the h-reduced graph G_h. First, for each large-vertex v_i, it is always possible to find $\deg(v_i) - 2k$ edges of the form $[v_i, w_i]$, where w_i is a small-vertex. This is because v_i has at least $2k$ neighbors while there are only $h < k$ large-vertices. Secondly, since we only delete edges whose one end is a large-vertex and the other end is a small-vertex, when we delete edges incident to a large-vertex, no other large-vertices would change their degrees. In particular, all large-vertices in the h-reduced graph G_h have degree exactly $2k$.

Lemma 3.2. *Let G be a graph that has h large-vertices $v_1, v_2, \ldots, v_h$, with $h < k$, and let G_h be an h-reduced graph of G. Then the graph G has a k-matching if and only if the h-reduced graph G_h has a k-matching.*

Proof. Since the h-reduced graph G_h is a subgraph of the graph G, if G_h has a k-matching, then obviously the graph G has a k-matching.

To prove the other direction, assume, to the contrary, that the graph G has a k-matching but the h-reduced graph G_h has no k-matching. Suppose that a k-matching in G can have at most r edges in the h-reduced graph G_h. Thus, $r < k$. Let $\mathcal{A}_r$ be the set of all k-matchings in G that have exactly r edges in the h-reduced graph G_h. We first study the properties of k-matchings in the set $\mathcal{A}_r$. Let M be any k-matching in $\mathcal{A}_r$. Since the graph G_h has no k-matching, there is at least one edge e_0 in M that is not in G_h.

(1) The k-matching M must cover all large-vertices. To see this, suppose that M does not cover a large-vertex v_i. Consider the $(k-1)$-matching $M_0^- = M \setminus \{e_0\}$, where e_0 is an edge in M that is not in G_h. The $(k-1)$-matching M_0^- also contains r edges in the h-reduced graph G_h. There are at most $2k-2$ neighbors of the large-vertex v_i in G_h that are covered by the $(k-1)$-matching M_0^-. Since the large-vertex v_i has $2k$ neighbors in G_h, there is a neighbor w_i of v_i in G_h that is not covered by M_0^-. Therefore, $M_0^- \cup \{[v_i, w_i]\}$ gives a k-matching in G that has $r+1$ edges in G_h, contradicting the assumption that a k-matching in G can have at most r edges in G_h. This contradiction proves that the k-matching M must cover all large-vertices.

(2) The k-matching M does not contain edges whose both ends are large-vertices. Suppose that M contains an edge $e_1 = [v_i, v_j]$ whose both ends v_i and v_j are large-vertices. First note that the edge e_1 must be in the graph G_h since in the construction of the h-reduced graph G_h, we never delete edges whose both ends are large-vertices. Since at most $2k-2$ neighbors of the large-vertex v_i can be covered by the $(k-1)$-matching $M_1^- = M \setminus \{e_1\}$ and since the large-vertex v_i has $2k$ neighbors in G_h, at least one neighbor $w_i \neq v_j$ of v_i in G_h is not covered by M_1^-. Thus, replacing the edge $e_1 = [v_i, v_j]$ by the edge $[v_i, w_i]$ gives a k-matching that has r edges in G_h but leaves the large-vertex v_j uncovered. But this contradicts what we have proved in (1) that a k-matching in $\mathcal{A}_r$ must cover all large-vertices.

(3) The k-matching M cannot contain an edge $e_2 = [v_i, x_i]$ in G that is not in the h-reduced graph G_h, where v_i is a large-vertex. Again if such an edge e_2 exists, then there must be a neighbor w_i of v_i in G_h such that w_i is not covered by the $(k-1)$-matching $M_2^- = M \setminus \{e_2\}$. Thus, the k-matching $M_2^- \cup \{[v_i, w_i]\}$ would give a k-matching in G that has $r+1$ edges in the h-reduced graph G_h, contradicting the definition of r.

Summarizing (1)-(3), we conclude that the k-matching M must contain h edges in the h-reduced graph G_h, with one end being a large-vertex and the other end being a small-vertex. Since there are only h large-vertices in the graph G, the other $k-h$ edges in M must have their both ends being small-vertices. Because in the construction of the h-reduced graph G_h, we never delete edges whose both ends are small-vertices, these $k-h$ edges in M must also be in the h-reduced graph G_h. Thus, the k-matching M in G is a k-matching in the h-reduced graph G_h, contradicting the assumption that G_h has no k-matching, thus, proving the lemma. □

Algorithm UGM

INPUT: an unweighted graph G and parameter k

OUTPUT: a k-matching in G, or report no such a matching in G.

1. collect up to k large-vertices in G, store them in V_L; let $h = |V_L|$;
2. **if** $(h = k)$ **return** a k-matching M in G;
3. construct the h-reduced graph G_h but keep up to $(4k-3)(k-h)$ small-edges;
4. **if** (G_h has $(4k-3)(k-h)$ small-edges) **return** a k-matching M in G;
5. call Best-Match(G_h) to solve the problem.

Fig. 1. The k-matching algorithm for unweighted graphs.

By Lemma 3.2, it suffices to consider how to construct a k-matching in the h-reduced graph G_h. Unfortunately, because of the space limit, we cannot construct the h-reduced graph explicitly and store it in the working memory: the number of edges whose both ends are small-vertices in the original G can be as large as $\Omega(N)$, which are all contained in the h-reduced graph. In the following, we show how we can construct a k-matching in an "implicit" h-reduced graph G_h. For simplicity, we will call an edge e a *small-edge* if both ends of e are small-vertices.

Lemma 3.3. *Let $h < k$. If the h-reduced subgraph G_h has a subgraph G'_h that contains all edges that are incident to the large-vertices in G_h plus $(4k-3)(k-h)$ small-edges in G_h, then G'_h has a k-matching that can be constructed in time $O(k^2 \log k)$ and space $O(k^2)$.*

Proof. First note that the graph G'_h can be stored in space $O(k^2)$. We construct a k-matching in the graph G'_h, as follows: (1) start with an empty matching M; and (2) repeatedly pick an edge e from the remaining small-edges, include e in the matching M, and delete the two ends of e (and all incident edges). Since there are at most $4k-4$ other small-edges that can share common ends with e, with the $(4k-3)(k-h)$ small-edges in G'_h, we will be able to construct a matching of $k-h$ edges in G'_h. Now, as we did in Lemma 3.1, we proceed with each v_i of the h large-vertices $\{v_1, \ldots, v_h\}$, where we can find an edge $[v_i, w_i]$, where w_i is a small-vertex not covered by M, so we can add the edge $[v_i, w_i]$ to the matching M. This gives a k-matching M in the graph G'_h.

To achieve the time complexity given in the lemma, we store the edges and vertices of the graph G'_h in balanced search trees so that searching, insertion, and deletion take $O(\log k)$ time per operation, which leads to the $O(k^2 \log k)$ running time of the algorithm. □

Now we are ready for our matching algorithm for unweighted graphs, as given in Fig. 1, where Best-Match in step 5 is an algorithm that solves the k-matching problem in the h-reduced subgraph G_h, whose complexity will be discussed in detail later. In order to keep the running time of the algorithm **UGM** to be linear in terms of the input size N, we need to use certain randomness, which will be explained in the proof of Theorem 3.4. Thus, our algorithm is a randomized algorithm, whose error bound and complexity are given in the following theorem.

Theorem 3.4. *For any $\epsilon > 0$, with probability at least $1-\epsilon$, the algorithm* **UGM** *solves the p-UGM problem in time $O(N + k^2 \log k + k \log(1/\epsilon) + \alpha(k^2))$ and space $O(k^2)$, where $\alpha(k^2)$ is the time complexity for finding a k-matching in a graph of $O(k^2)$ edges and without degree-0 vertices, with the space complexity simultaneously bounded by $O(k^2)$.*

Proof. The correctness of the algorithm is obvious: Lemma 3.1 and Lemma 3.3 ensure, respectively, that if the algorithm returns at step 2 and step 4, then it returns a k-matching in the graph G. If the algorithm returns from step 5, then Lemma 3.2 guarantees that the algorithm returns a k-matching in the h-reduced graph G_h, which is also a k-matching in the original graph G, if and only if the original graph G has a k-matching.

We analyze the complexity of the algorithm. Recall that the graph G is given in an adjacency list. Thus, the degree of a vertex can be computed by reading the list of neighbors of the vertex. As a result, step 1 takes time $O(N)$. Since we keep at most k large-vertices of G in the set V_L, the set V_L can be stored in space $O(k)$. In case the number h of large-vertices in the set V_L is equal to k, by Lemma 3.1, step 2 of the algorithm constructs and returns a k-matching M in G in time $O(N + k^2 \log k)$ and space $O(k)$.

If the number h of large-vertices in the set V_L is smaller than k, then step 3 of the algorithm constructs the h-reduced graph G_h. For this, we need to be more careful: in order to collect the small-edges, we need to decide for each edge if any endpoint of the edge is a large-vertex. Even if we organize the large-vertices in a balanced search tree, it would still take time $O(N \log h) = O(N \log k)$ to go through the edges of G and construct the h-reduced graph G_h.

To solve this problem, we use the technique of universal hashing [9]. For an integer $m \geq 1$, denote by $[m]$ the integer set $\{1, 2, \ldots, m\}$. For the set V_L of the h collected large-vertices, we pick a hash function H from U to $[h^2]$ randomly from a universal class of hash functions, where U is the set of the vertices in the input graph G. With probability at least $1/2$, the function H is injective from the set V_L to $[h^2]$ (see [9], Theorem 11.9). The hash function H can be constructed in constant

(randomized) time (see [9], Theorem 11.5). Moreover, after initializing an array $A[1..h^2]$ in time $O(h^2)$, we can check if the function H is injective from V_L in time $O(h)$: for this, we fix a distinct value a_H for the function H, and for each vertex v in V_L, if $A[H[v]]$ is already equal to a_H, then the function H is not injective from V_L, otherwise, we set $A[H[v]] = a_H$. Therefore, for any $\epsilon > 0$, by repeating this procedure $\log(1/\epsilon)$ times, thus in time $O(\log(1/\epsilon)h + h^2)$ (note that the array $A[1..h^2]$ needs to be initialized only once), with probability at least $1-\epsilon$, we will get a hash function H_0 that is injective from V_L. Using this hash function H_0, we re-initialize the array $A[1..h^2]$, and then place the h large-vertices in V_L in the array $A[1..h^2]$ so that a large-vertex w is placed in $A[H_0[w]]$. Since $h < k$, we conclude that with probability at least $1-\epsilon$ and in time $O(\log(1/\epsilon)k + k^2)$ and space $O(k^2)$ (which is mainly for the array $A[1..h^2]$), we will find the hash function H_0 that is injective from V_L and finalize the array $A[1..h^2]$. Now for any vertex v in the input graph G, by checking the value $A[H_0(v)]$, which takes constant time, we can easily find out if v is a large-vertex.

Now it is straightforward to construct the h-reduced graph G_h. We simply scan the input graph G. For each large-vertex v_i, we delete all but $2k$ edges incident to v_i, (keeping all the edges of the form $[v_i, v_j]$ where v_j is also a large-vertex). For each small-vertex w, we record the small-edges incident to w. The process stops either when we have collected $(4k-3)(k-h)$ small-edges, or when all edges of the graph G are examined. In the former case, we get a subgraph G'_h of the h-reduced graph G_h that satisfies the conditions of Lemma 3.3, thus, step 4 of the algorithm **UGM** constructs a k-matching M in G'_h (thus also in G_h and in G) in time $O(k^2 \log k)$ and space $O(k^2)$. In the latter case, the h-reduced graph G_h has fewer than $2kh + (4k-3)(k-h) = O(k^2)$ edges, so the algorithm Best-Match in step 5 is applied on the graph G_h with $O(k^2)$ edges and Lemma 3.2 guarantees the correctness of the algorithm **UGM**.

We remark that in this process, the vertices in G that become of degree-0 after the construction of the h-reduced subgraph G_h can also be efficiently identified and deleted: for each small-vertex w, we do not record *any* of its incident edges whose other end is a large-vertex. In particular, small-vertices in G that are adjacent to only large-vertices are not recorded in this scanning phase. Only after this scanning phase, we re-examine the chosen edges incident to large-vertices in the h-reduced graph G_h, and add further small-vertices to G_h if they are the other ends of these edges and are not recorded in the scanning phase. This prevents the graph G_h from having degree-0 vertices. Thus, the h-reduced graph G_h in step 5 has $O(k^2)$ edges and has no vertices of degree 0. As a result, the number n_h of vertices in the h-reduced subgraph G_h is also bounded by $O(k^2)$. Now we rename the vertices of G_h as integers in $[1..n_h]$ so that the Best-Match algorithm in step 5 can be applied. This takes another $O(k^2 \log k)$ time and space $O(k^2)$. By the assumption, the p-UGM problem on the graph G_h (thus by Lemma 3.2 on the input graph G) can be solved in time $\alpha(k^2)$ and space $O(k^2)$.

Summarizing all the above discussions completes the proof of the lemma. □

Now we study the time complexity $\alpha(k^2)$ of solving the k-matching problem in a graph with $O(k^2)$ edges (we will assume, without loss of generality, that graphs have no degree-0 vertices). There has been extensive research on algorithms for constructing a maximum matching in an unweighted graph [22,26,32]. In particular, it is known [26] that for a graph of n vertices and m edges, a maximum matching in the graph can be constructed in time $O(m\sqrt{n})$, from which the k-matching problem can be solved trivially. Therefore, for graphs of $O(k^2)$ edges, which may have up to $O(k^2)$ vertices, the k-matching problem can be solved in time $O(k^3)$, giving an upper bound $O(k^3)$ for the complexity $\alpha(k^2)$. In the following, we show how a better upper bound for the time complexity $\alpha(k^2)$ can be obtained.

Let M be a matching in a graph G. An *augmenting path* P (relative to M) in G is a simple path whose both ends are uncovered by M, and whose edges are alternatively going between not in M and in M. An augmenting path is the *shortest* if its length (i.e., the number of edges) is the minimum over all augmenting paths relative to M.

We start with the following theorem, which is also of its independent interests.

Theorem 3.5. *There is an $O(m\sqrt{k})$-time and $O(m)$-space algorithm that on a graph G of m edges, either constructs a k-matching in G or reports that no k-matching exists in G.*

Proof. We first prove the following claim:

Claim. *Let G be a graph of m edges, and let k_0 be the size of a maximum matching in G. A maximum matching in the graph G can be constructed in time $O(m\sqrt{k_0})$ and space $O(m)$.*

Proof of the Claim. An algorithm proposed by Micali and Vazirani [26] constructs a maximum matching in a general unweighted graph G of n vertices and m edges in time $O(m\sqrt{n})$ and space $O(m)$ (we will call this algorithm the *MV-algorithm*). The MV-algorithm runs in phases. Each phase starts with a matching M, finds a maximal set of vertex-disjoint shortest augmenting paths relative to M, and augments along all these paths to get a larger matching. As proved by Hopcroft and Karp (Theorem 3 in [22]), running the MV-algorithm for at most $2\sqrt{k_0}+1$ such phases will be sufficient to find a maximum matching in the graph G. Moreover, Micali and Vazirani [26] presented an $O(m)$-time algorithm (thus also in space $O(m)$) that implements the process of each phase in the MV-algorithm.[2] Combining these two results, we obtain an algorithm that

[2] This $O(m)$-time algorithm for each phase in the MV-algorithm is highly nontrivial. For much more details and discussions, see [32,33] On the other hand, for bipartite graphs, there is a much simpler $O(m)$-time algorithm that implements the process of each phase. See [22].

finds a maximum matching in a general unweighted graph G of m edges in time $O(m\sqrt{k_0})$ and space $O(m)$. This proves the claim.

Let us now get back to the proof of the original theorem. Our algorithm proceeds as follows. We first use a trivial greedy algorithm to construct a maximal matching M' in the graph G in time $O(m)$ and space $O(m)$. If $|M'| \geq k$, then we can easily have a k-matching of G from M'. On the other hand, we have $|M'| < k$. It is well-known that for a graph the size of a maximum matching is at most twice of that of a maximal matching [30]. Therefore, if $|M'| < k$, then the maximum matching in the graph G has its size k_0 bounded by $2k$, and we can apply the above claim to construct a maximum matching M'' in G in time $O(m\sqrt{2k}) = O(m\sqrt{k})$ and space $O(m)$. Now from the maximum matching M'', we can easily either construct a k-matching in G or report that the graph G has no k-matching. This proves the theorem. □

By Theorem 3.5, we get an upper bound $O(k^{2.5})$ on the time complexity $\alpha(k^2)$ given in Theorem 3.4 for the algorithm **UGM**. Now if we replace $\alpha(k^2)$ with $k^{2.5}$, and let $\epsilon = 1/2^{k^{1.5}}$, then Theorem 3.4 reads as

Theorem 3.6. *With probability at least* $1 - 1/2^{k^{1.5}}$*, the algorithm* **UGM** *solves the p-UGM problem on general unweighted graphs in time* $O(N + k^{2.5})$ *and space* $O(k^2)$.

Note that the bound $O(N + k^{2.5})$ in Theorem 3.6 is the best possible for the p-UGM problem based on the current status of the research on graph matching algorithms – the best known algorithm for the graph matching problem runs in time $O(n^{2.5})$ on a graph of n vertices [26].[3]

We are not aware of any parameterized algorithms published in the literature that are specifically for solving the p-UGM problem. On the other hand, in the research on streaming algorithms, the p-UGM problem and related problems have been studied recently. Chitnis et al. [7] proposed a deterministic algorithm on the insert-only streaming model for the parameterized Vertex-Cover problem. The algorithm is based on an algorithm on the insert-only streaming model, running in space $O(k^2)$ and update time $O(\log k)$, which constructs a *maximal* matching of up to k edges (thus, *not* solving the problem p-UGM). This algorithm, if measured on our model, is a deterministic algorithm of time $O(N \log k)$ and space $O(k^2)$ that constructs a maximal matching of up to k edges. Later, Chitnis et al. [5] presented two randomized algorithms for the p-UGM problem on the dynamic streaming model. In order to deal with edge deletions in streaming, the algorithms given in [5] smartly employed powerful techniques in l_0-sampling [10]. However, these techniques are relatively expensive. If we remove these expensive operations, the algorithms given in [5] can be used to solve the p-UGM problem (in the insert-only graph streaming model). With the simplifications, the first algorithm given in [5], for any $\epsilon > 0$, runs in time $O(N \log(1/\epsilon) + \beta(k))$ and converts an input graph G of size N into a graph of up to $O(k^4 \log(1/\epsilon))$ edges with a probability $1 - \epsilon$. Thus, both the bound $\beta(k)$ in the time complexity and that in the space complexity of the algorithm are at least $O(k^4 \log(1/\epsilon))$. Moreover, to achieve a probability $1 - o(1)$, the algorithm would require super-linear time. The second algorithm given in [5], if simplified as described above, converts a graph G of size N into a graph with $O(k^2 \log(1/\epsilon))$ edges. The algorithm runs in super-linear time $O(N \log k)$ (and super-quadratic space) if we want to achieve a success probability $1 - 1/k^{O(1)}$. More seriously, the algorithm only applies to graphs in which the size of a maximum matching is bounded by $O(k)$. In comparison, our algorithm given in Theorem 3.6 runs in space $O(k^2)$ and in "strong" linear time in terms of the input graph size N: the term N in the time complexity of our algorithm is independent of the parameter k and of the success probability. Moreover, our algorithm assumes no constraints on the size of a maximum matching of the input graph and has a much higher success probability $1 - 1/2^{k^{1.5}}$.

4. Case study II: matching in weighted graphs

In this section, we study the Maximum Weighted k-Matching problem on weighted graphs, i.e., the *p-WGM problem*. Let G be an (edge-)weighted graph (or, simply, a *weighted graph*). A *maximum k-matching* in G is a k-matching in G whose weight is the largest over all k-matchings in G. The instances of the p-WGM problem consist of pairs of the form (G, k), where G is a weighted graph and k is an integer. A solution to the instance (G, k) is either a maximum k-matching in G or a report that no k-matching exists in G.

We remark that in practice, the p-WGM problem is probably applicable to more applications, compared to the p-UGM problem. Indeed, with a very large graph G, we may only be interested in having a certain number k of matched vertex pairs where k is not necessarily the largest. On the other hand, we may want to have k such matched pairs that maximize an objective value.

Technically, the p-WGM problem becomes very different from the p-UGM problem. A weighted graph G may have matchings of very large size (the *size* of a matching is the number of edges in the matching) while we are just looking for a

[3] We remark that there is a randomized algorithm of time $O(n^{2.376})$ for the graph matching problem, based on fast matrix multiplication algorithms [27]. However, our h-reduced subgraph G_h may have up to $\Omega(k^2)$ vertices. Therefore, a direct application of the algorithm in [27] would not lead to a faster algorithm for the p-UGM problem. Moreover, using the algorithm in [27] would require space $O(k^4)$.

k-matching of the maximum weight in the graph, where k could be relatively small. In particular, Lemmas 3.1-3.3 are no longer useful in constructing maximum k-matchings: both Lemmas 3.1 and 3.3 construct k-matchings using edges incident to large-vertices. However, for weighted graphs, there is no guarantee that the edges incident to large-vertices are contained in maximum k-matchings. Finally, the technique we used in the proof of Theorem 3.4 to pre-scan the graph G and collect the large-vertices cannot be used – there can be simply too many large-vertices for our limited local space.

We start with the following lemma that will be useful in several places in our construction.

Lemma 4.1. *There is an algorithm that on an input of n elements and a parameter k, produces the k largest elements in the input in time $O(n)$ and space $O(k)$.*

Proof. The algorithm starts by reading the first k elements $\{a_1, a_2, \ldots, a_k\}$ from the input. Inductively, suppose that for an integer $i \geq k$, the algorithm has obtained the k largest elements b_1, b_2, ..., b_k in the first i elements in the input. The algorithm then reads the next block $\{a_{i+1}, a_{i+2}, \ldots, a_{i+k}\}$ of k elements in the input, and uses the linear-time Median-Finding algorithm [9] to find the k-th largest element in the set $S_{i+k} = \{b_1, b_2, \ldots, b_k, a_{i+1}, a_{i+2}, \ldots, a_{i+k}\}$ in time $O(k)$, from which the k largest elements in the set S_{i+k}, which are also the k largest elements in the first $i+k$ elements in the input, can be easily obtained. Since the algorithm spends time $O(k)$ on each block of k elements in the input, we conclude that the running time of the algorithm is $O(n)$. Moreover, it is obvious that the algorithm takes $O(k)$ space. □

Let G be a weighted graph. Similarly (but not identically) to the process on the problem p-UGM, we define a *large-vertex* to be a vertex whose degree is at least $8k$ and a *small-vertex* to be a vertex whose degree is less than $8k$. In the following, we will introduce operations that remove edges from the weighted graph G without changing the weight of its maximum k-matchings. This will require the condition that each edge in the weighted graph G has a distinct weight, which, in general, is not the case. For this, we introduce a new edge weight function for the graph G as follows: let $e = [v, w]$ be an edge of weight $wt(e)$ in the graph G, we define the new weight $wt'(e)$ for the edge e as a triple $wt'(e) = (wt(e), \min\{v, w\}, \max\{v, w\})$. The new edge weights follow the lexicographic order. In terms of the weight function $wt'(\,)$, each edge in the graph G has a distinct weight. Moreover, for any edge set S and any integer h, the set of the h heaviest edges in S in terms of the weight function $wt'(\,)$, which is uniquely defined, must be a set that consists of h heaviest edges in S in terms of the weight function $wt(\,)$.

We first consider the following two kinds of subgraphs constructed from the weighted graph G, where the edge weights are measured by the new edge weights $wt'(\cdot)$ as defined above:

- The *trimmed subgraph* G_T of the graph G consists of the edges $e = [v, w]$ in G such that e is among the $8k$ heaviest edges incident to the vertex v and among the $8k$ heaviest edges incident to the vertex w, plus the vertices incident to these edges.
- The *reduced subgraph* G_R of G is a subgraph of the trimmed subgraph G_T of G such that either $G_R = G_T$ if G_T has no more than $k(16k-1)$ edges, or G_R consists of the $k(16k-1)$ heaviest edges in G_T, plus the vertices incident to the edges.

Remark 1. Note that every edge incident to a small-vertex v is among the $8k$ heaviest edges incident to the vertex v.

Remark 2. Because each edge e in the graph G has a distinct edge weight $wt'(e)$, the trimmed subgraph G_T and the reduced subgraph G_R of the graph G are uniquely defined.

Remark 3. Each vertex in the trimmed subgraph G_T, thus also each vertex in the reduced subgraph G_R, has degree bounded by $8k$. Note that a large-vertex v in the graph G may have degree less than $8k$ in the trimmed subgraph G_T. In particular, if an edge $e = [v, w]$ is among the $8k$ heaviest edges incident to v but not among the $8k$ heaviest edges incident to w, then the degree of the vertex v in the trimmed subgraph G_T is less than $8k$.

Remark 4. The size of the trimmed subgraph G_T can still be very large (since there can be many large-vertices). On the other hand, the reduced subgraph G_R has size bounded by $O(k^2)$.

Lemma 4.2. *If a weighted graph G has k-matchings, then its trimmed subgraph G_T also has k-matchings. Moreover, a maximum k-matching in the trimmed subgraph G_T is also a maximum k-matching in the original graph G.*[4]

Proof. For each large-vertex v in the graph G, let $e_{8k}(v)$ be the $(8k)$-th heaviest edge incident to v. Consider the algorithm in Fig. 2 that constructs the trimmed subgraph G_T.

[4] Note that although when we compare edges we use the new weight function $wt'(\,)$, the weight of a matching is still defined in terms of the original edge weight function $wt(\,)$.

Algorithm Trimming

1. sort the large-vertices in the weighted graph G in a sequence: $v'_1, v'_2, \ldots, v'_h$, such that $wt'(e_{8k}(v'_1)) \leq wt'(e_{8k}(v'_2)) \leq \cdots \leq wt'(e_{8k}(v'_h))$;
2. **for** $i = 1$ **to** h **do** delete all but the $8k$ heaviest edges incident to v'_i.

Fig. 2. Constructing the trimmed subgraph G_T of a weighted graph G.

Since every edge e in the graph G has a distinct edge weight $wt'(e)$, when we delete edges incident to a large-vertex v'_i, we would not delete any of the $8k$ heaviest edges incident to a large-vertex v'_j with $i < j$. Therefore, if we let G_i be the graph G after deleting all but the $8k$ heaviest edges incident to the vertex v'_s for all $s \leq i$, then the graph G_{i+1} will be obtained from the graph G_i by deleting all but the $8k$ heaviest edges incident to the vertex v'_{i+1}, and the graph G_h constructed by the algorithm is the trimmed subgraph G_T. Now suppose that the weighted graph G has k-matchings. We prove by induction on i that for all i, the graph G_i has k-matchings, and a maximum k-matching in G_i is also a maximum k-matching in the graph G. This is certainly true for $i = 0$.

Inductively, assume that the graph G_i has k-matchings, and let M_i be a maximum k-matching in G_i, which is also a maximum k-matching in G. Consider the graph G_{i+1} that is obtained from G_i by deleting all but the $8k$ heaviest edges incident to the vertex v'_{i+1}. If M_i contains no edge that is deleted in the construction of G_{i+1} from G_i, then M_i is also a k-matching in G_{i+1}, thus is a maximum k-matching in G_{i+1}. Otherwise, M_i contains an edge $[v'_{i+1}, w]$ that is not among the $8k$ heaviest edges $[v'_{i+1}, w_s]$, $1 \leq s \leq 8k$, incident to the vertex v'_{i+1} in the graph G_i. Since the $(k-1)$-matching $M_i \setminus \{[v'_{i+1}, w]\}$ can cover at most $2k-2$ neighbors of v'_{i+1}, there must be an edge $[v'_{i+1}, w_t]$ among the $8k$ heaviest edges incident to v'_{i+1} such that the vertex w_t is not covered by $M_i \setminus \{[v'_{i+1}, w]\}$. Thus, replacing the edge $[v'_{i+1}, w]$ with the edge $[v'_{i+1}, w_t]$ will give a k-matching M_{i+1} in the graph G_{i+1}. Moreover, by the definition of the weight $wt'(\cdot)$, the k-matching M_{i+1} in G_{i+1} has a weight at least as large as that of the maximum k-matching M_i in the original graph G. Since G_{i+1} is a subgraph of G, we conclude that M_{i+1} is a maximum k-matching in G. This proves that the graph G_{i+1} also has k-matchings and a maximum k-matching in G_{i+1} is also a maximum k-matching in the original graph G. Letting $i = h$ and noting that G_h is the trimmed subgraph G_T complete the proof of the lemma. □

Lemma 4.3. *If a weighted graph G has k-matchings, then its reduced subgraph G_R also has k-matchings. Moreover, a maximum k-matching in the reduced subgraph G_R is also a maximum k-matching in the original graph G.*

Proof. By Lemma 4.2, it suffices to prove that if the trimmed subgraph G_T has k-matchings, then the reduced subgraph G_R also has k-matchings and that a maximum k-matching in the reduced subgraph G_R is also a maximum k-matching in the trimmed subgraph G_T. If the trimmed subgraph G_T has fewer than $k(16k-1)$ edges, then by definition, $G_R = G_T$, and the lemma obviously holds true. Thus, we can assume that the reduced subgraph G_R has exactly $k(16k-1)$ edges, which are the $k(16k-1)$ heaviest edges in the trimmed subgraph G_T. Now suppose that the trimmed subgraph G_T has k-matchings and let M_T be a maximum k-matching in G_T. Assume $M_T = M'_T \cup M''_T$, where M'_T is the set of edges that are in the reduced subgraph G_R and M''_T is the set of edges that are not in the reduced subgraph G_R, with $|M'_T| = h$ and $|M''_T| = k - h > 0$. Now for each edge e in M'_T, delete the two ends of e (and all incident edges) in the graph G_R. Since the graph G_R has $k(16k-1)$ edges, and the vertex degree of G_R is bounded by $8k$, this will delete at most $h(16k-1)$ edges in G_R. Thus, the resulting graph G'_R still has at least $k(16k-1) - h(16k-1) = (k-h)(16k-1)$ edges. Now in the graph G'_R, because the vertex degree is bounded by $8k$, we can easily construct a $(k-h)$-matching M''_R in G'_R (thus in G_R) by repeatedly including an (arbitrary) edge in the matching and removing all edges incident to the ends of the edge. Since no edge in M''_T is in G_R, by the definition of the reduced subgraph G_R, the weight of the $(k-h)$-matching M''_R in G_R is at least as large as that of the $(k-h)$-matching M''_T in G_T. Therefore, replacing the $(k-h)$-matching M''_T in M_T with the $(k-h)$-matching M''_R gives a k-matching $M'_T \cup M''_R$ in the reduced subgraph G_R whose weight is at least as large as that of the maximum k-matching M_T in the trimmed subgraph G_T. As a consequence, the weight of the maximum k-matching in the reduced subgraph G_R is at least as large as that of the maximum k-matching M_T in the trimmed subgraph G_T. Since G_R is a subgraph of G_T, we conclude that a maximum k-matching in the reduced subgraph G_R is also a maximum k-matching in the trimmed subgraph G_T. □

By Lemma 4.3, to construct a maximum k-matching in the input graph G, it suffices to construct a maximum k-matching in the reduced subgraph G_R, which is a subgraph of the trimmed subgraph G_T and has a size $O(k^2)$. However, it seems challenging to construct the reduced subgraph G_R from the weighted graph G in time $O(N + k^{O(1)})$ and space $k^{O(1)}$:

(1) The trimmed subgraph G_T can be very large, and we may not have enough space to store the entire trimmed subgraph G_T;
(2) The number of large-vertices can be very large. Although any proper subset of at least k large-vertices and their incident edges contain a k-matching in G, there is no guarantee that the k-matching is of the maximum weight. On the other hand, we may not have enough space to record all large-vertices;

Algorithm RSubG

INPUT: a weighted graph G and parameter k

OUTPUT: the reduced subgraph G_R of G.

1. construct the bounding set B_{8k} of large-vertices in G;
   for (each $v \in B_{8k}$) construct the bounding list L_v^{8k} for the vertex v;
2. construct an injective hash function H from B_{8k} to $[(8k)^2]$;
3. let E_R be the set of edges in G_T that are in the set $\bigcup_{v \in B_{8k}} L_v^{8k}$;
4. **if** $(|E_R| < k(16k-1))$
4.1 **for** (each vertex v in G such that $v \notin B_{8k}$)
4.2 **for** (each edge $e = [v, w]$ in L_v^{8k} such that $w \notin B_{8k}$)
add e to E_R but only keep the $k(16k-1)$ heaviest edges in E_R;
5. **else** \\ $|E_R| \geq k(16k-1)$
5.1 delete all but the $k(16k-1)$ heaviest edges in E_R;
let $m_0 = \min\{wt'(e) \mid e \in E_R\}$;
5.2 **for** (each vertex v in G such that $v \notin B_{8k}$)
5.3 **for** (each edge $e = [v, w]$ in L_v^{8k} such that $w \notin B_{8k}$)
5.4 **if** $(wt'(e) \geq m_0)$
add e to E_R but only keep the $k(16k-1)$ heaviest edges in E_R;
6. let $G_R = (V_R, E_R)$, where V_R is the set of vertices incident to edges in E_R.

Fig. 3. Constructing the reduced subgraph G_R of a weighted graph G.

(3) Because of (2), even constructing the trimmed subgraph G_T "locally" becomes difficult: to determine if an edge $e = [v, w]$ of G is in G_T, we need to know if $wt'(e) \geq wt'(e_{8k}(v))$ and $wt'(e) \geq wt'(e_{8k}(w))$. Note that this should be done in constant time in average, in order to achieve the $O(N + k^{O(1)})$ time complexity for the construction of the reduced subgraph G_R;
(4) In order to keep the size of the reduced subgraph G_R by $O(k^2)$, we also need to exclude the vertices of G that are incident to no edges in G_R.

We develop new techniques to deal with these technical difficulties. Again for a large-vertex v, we let $e_{8k}(v)$ be the $(8k)$-th heaviest edge incident to v in the graph G, in terms of the weight function $wt'(\,)$. The value $wt'(e_{8k}(v))$ will be called the *e_{8k}-value* of the vertex v. For the convenience of discussions, we define the e_{8k}-value of a small-vertex to be $-\infty$.

The *bounding set* B_{8k} of large-vertices in the graph G is defined as follows:

(1) if there are at most $8k$ large-vertices in G, then B_{8k} contains all large-vertices; and
(2) if there are more than $8k$ large-vertices in G, then B_{8k} contains the $8k$ large-vertices whose e_{8k}-values are among the $8k$ largest e_{8k}-values over all large-vertices of G.

Similarly, for a vertex v, we define the *bounding list* L_v^{8k} of edges incident to v as follows:

(1) if v is a small-vertex, then L_v^{8k} consists of all edges incident to v; and
(2) if v is a large vertex, then L_v^{8k} consists of the $8k$ heaviest edges incident to v.

Our algorithm that constructs the reduced subgraph G_R of the graph G is presented in Fig. 3. Recall that by $[m]$, we denote the integer set $\{1, 2, \ldots, m\}$.

We first prove the correctness of the algorithm **RSubG** given in Fig. 3.

Lemma 4.4. *The algorithm* **RSubG** *given in Fig. 3 constructs the reduced subgraph G_R of the weighted graph G.*

Proof. We start with the following observations:

Claim 1. *If the bounding set B_{8k} in step 1 of the algorithm* ***RSubG*** *contains $8k$ vertices, then the edge set E_R in step 3 contains more than $k(16k-1)$ edges.*

Proof of Claim 1. Under the condition of the claim, place the $8k$ large-vertices in B_{8k} into an ordered list $B'_{8k} = (v_1, v_2, \ldots, v_{8k})$, where all v_i are large-vertices in G whose e_{8k}-values are among the $8k$ largest e_{8k}-values over all large-vertices in G, and the vertices in the list B'_{8k} are sorted non-decreasingly in terms of their e_{8k}-values. For any vertex v_i in the list B'_{8k}, let $e = [v_i, w]$ be an edge incident to v_i, where w is either a vertex v_j in the list B'_{8k} with $j < i$ or a vertex

not in the list B'_{8k}. By definition, we have $wt'(e_{8k}(w)) \le wt'(e_{8k}(v_i))$. Therefore, if $e \in L^{8k}_{v_i}$, i.e., if e is among the $8k$ heaviest edges incident to v_i, then e is also among the $8k$ heaviest edges incident to w, i.e., $e \in L^{8k}_{w}$, which means that the edge e is in the trimmed subgraph G_T. As a result, among the $8k$ heaviest edges incident to the vertex v_i in the list B'_{8k}, only those that are between v_i and v_p, where v_p is a vertex in B'_{8k} with $i < p$, can be missing in the trimmed subgraph G_T. Thus, there are at least i edges incident to the vertex v_i in the trimmed subgraph G_T, i.e., the degree of the vertex v_i in the trimmed subgraph G_T is at least i. Let G^{8k}_T be the graph that consists of the edges that are both in the trimmed subgraph G_T and in the set $\bigcup_{v\in B_{8k}} L^{8k}_v$, then the degree sum of the vertices in G^{8k}_T is at least $\sum_{i=1}^{8k} i = 4k(8k+1)$, which implies that the number of edges in the graph G^{8k}_T (i.e., the number of edges in the set E_R) is at least $4k(8k+1)/2 > k(16k-1)$. This completes the proof of the claim.

Claim 1 directly implies the following result:

Claim 2. *If the condition $|E_R| < k(16-1)$ in step 4 of the algorithm **RSubG** holds, then the graph constructed in step 6 is the reduced subgraph G_R of the graph G.*

Proof of Claim 2. If $|E_R| < k(16-1)$ in step 4, then by Claim 1, the set B_{8k} contains fewer than $8k$ vertices, which implies that *all* large-vertices of the graph G are included in the set B_{8k}, and the set E_R constructed in step 3 contains *all* edges in the trimmed subgraph G_T that are incident to *any* large-vertices in G. Therefore, the only edges in G_T that are missing in the set E_R are the edges whose both ends are small-vertices in G, i.e., vertices that are not in the set B_{8k} (note that these edges are all in the trimmed subgraph G_T). Now steps 4.1-4.2 go through exactly all these edges and, together with the edges of G_T that are already in the set E_R after step 3, record the (up to) $k(16k-1)$ heaviest edges. By the definition, these are exactly the edges that make up the reduced subgraph G_R. This proves the claim.

The remaining case is that the set E_R contains at least $k(16k-1)$ edges after step 3. Note that in this case, there can be large-vertices that are not included in the bounding set B_{8k}. After step 5.1, the set E_R contains exactly $k(16k-1)$ edges, which are the $k(16k-1)$ heaviest edges among all edges in G_T that are incident to vertices in B_{8k}. By the definition of the reduced subgraph, the edges deleted from the set E_R in step 5.1 cannot be in the reduced subgraph G_R. Therefore, all edges in G_T that are in the set $\bigcup_{v\in B_{8k}} L^{8k}_v$ and can possibly be in the reduced subgraph G_R are included in the set E_R after step 5.1. As a result, the edges that can possibly be in the reduced subgraph and are not yet included in the set E_R after step 5.1 are those whose both ends are not in the set B_{8k}. Steps 5.2-5.3 examine all these edges.

Claim 3. *If the edge $e = [v, w]$ in step 5.3 of the algorithm **RSubG** satisfies $wt'(e) \ge m_0$, then the edge e is in the trimmed subgraph G_T.*

Proof of Claim 3. Let e_0 be the edge in the edge set $\bigcup_{v\in B_{8k}} L^{8k}_v$ that has the minimum edge weight, in terms of the edge weight function $wt'(\,)$. By the definition, e_0 must be the $(8k)$-th heaviest edge incident to a vertex v_i in B_{8k}. Thus, $wt'(e_0) = wt'(e_{8k}(v_i))$ is the e_{8k}-value of the vertex v_i in B_{8k}. Since the set E_R constructed in step 5.1 is a subset of the set $\bigcup_{v\in B_{8k}} L^{8k}_v$, we have $m_0 \ge wt'(e_{8k}(v_i))$. Now, for the edge $e = [v, w]$ in step 5.3, where both v and w are not in B_{8k}, by the definition of the set B_{8k}, we must have $wt'(e_{8k}(v)) \le wt'(e_{8k}(v_i))$ and $wt'(e_{8k}(w)) \le wt'(e_{8k}(v_i))$ (recall that the e_{8k}-value of a small-vertex is defined to be $-\infty$). Therefore, if the edge $e = [v, w]$ satisfies $wt'(e) \ge m_0$, then we must have $wt'(e) \ge wt'(e_{8k}(v))$ and $wt'(e) \ge wt'(e_{8k}(w))$, i.e., the edge e must be in the set intersection $L^{8k}_v \cap L^{8k}_w$, thus, in the trimmed subgraph G_T. This completes the proof of the claim.

The edge set E_R after step 5.1 contains exactly $k(16k-1)$ edges. By Claim 3, only edges in the trimmed subgraph G_T can be added to E_R, and the set E_R always contains exactly $k(16k-1)$ edges in the trimmed subgraph G_T.

Claim 4. *If the edge e in step 5.3 of the algorithm **RSubG** satisfies $wt'(e) < m_0$, then the edge e cannot be in the reduced subgraph G_R.*

Proof of Claim 4. If the edge e is not in the trimmed subgraph G_T, then of course e cannot be in the reduced subgraph G_R. Now suppose that e is in the trimmed subgraph G_T. By the way the set E_R is updated in step 5.4 and by Claim 3, the set E_R always contains exactly $k(16k-1)$ edges in G_T and the edge weight of any edge in E_R is not smaller than m_0. Therefore, if $wt'(e) < m_0$, then the edge e cannot be among the $k(16k-1)$ heaviest edges in the trimmed subgraph G_T, i.e., the edge e is not in the reduced subgraph G_R. The claim is proved.

Therefore, if the edge set E_R contains at least $k(16k-1)$ edges after step 3, which are the edges in both the trimmed subgraph G_T and the set $\bigcup_{v\in B_{8k}} L^{8k}_v$, then step 5.1 deletes from the set E_R some edges that obviously cannot be in the reduced subgraph G_R. Then, step 5.2-5.3 go through all edges that are not in the set $\bigcup_{v\in B_{8k}} L^{8k}_v$, ignore the edges that are obviously not in the reduced subgraph G_R (Claim 4), and examine the rest of the edges in the set in step 5.4 (by Claim 3,

all edges examined in step 5.4 are in the trimmed subgraph G_T). As a consequence, all edges in the trimmed subgraph G_T that are possibly in the reduced subgraph G_R are examined in steps 5.1-5.4. Since we only keep the $k(16k-1)$ heaviest such edges, we conclude that after step 5, the set E_R is the edge set of the reduced subgraph G_R. This gives us the following result:

Claim 5. *If the set E_R contains at least $k(16k-1)$ edges after step 3 of the algorithm **RSubG**, then the graph constructed in step 6 is the reduced subgraph G_R of the graph G.*

Combining Claim 2 and Claim 5 proves the lemma. □

Now we can draw a conclusion for the algorithm **RSubG** given in Fig. 3.

Lemma 4.5. *There is an algorithm such that, for any $\epsilon > 0$, with probability at least $1-\epsilon$, the algorithm on a weighted graph G of size N constructs the reduced subgraph G_R of G in time $O(N + k^2 + k\log(1/\epsilon))$ and space $O(k^2)$.*

Proof. By Lemma 4.4, it suffices to verify that the algorithm **RSubG** in Fig. 3 satisfies the probability requirement and the time and space complexities stated in the lemma.

To construct the bounding set B_{8k} in step 1, we scan the graph G. For each large-vertex v, we construct the bounding list L_v^{8k} as well as the e_{8k}-value for v. By Lemma 4.1, this will take time $O(\deg(v))$ and space $O(k)$, where the time complexity is, asymptotically, bounded by the amount of time for reading the edges incident to v. The e_{8k}-values of the large-vertices will be used as the keys in the construction of the bounding set B_{8k}. By Lemma 4.1, with additional $O(N)$ time and $O(k)$ space, we can construct the bounding set B_{8k}. Moreover, in this construction, we keep the bounding list L_v^{8k} for at most $O(k)$ vertices. Since the bounding set B_{8k} contains at most $8k$ vertices, the bounding set B_{8k} and the bounding lists for the vertices in the set B_{8k} can be constructed in time $O(N)$ and space $O(k^2)$ by step 1 of the algorithm.

Step 2 of the algorithm constructs a hash function H that is injective from the vertex set B_{8k} to $[(8k)^2]$, where the set B_{8k} contains at most $8k$ vertices. As we did for unweighted graphs in Theorem 3.4, a hash function that maps the set of vertices in the graph G to $[(8k)^2]$ and is randomly picked from a universal hashing class $\mathcal{H}$ has a probability at least $1/2$ to be injective from the set B_{8k} to $[(8k)^2]$ [9]. Therefore, with $\log(1/\epsilon)$-times of randomly picking a hash function from the universal hashing class $\mathcal{H}$, we will get a hashing function H that is injective from the set B_{8k} to $[(8k)^2]$, with probability at least $1-\epsilon$. Note that with an initiated array of size $(8k)^2$, we can easily verify in time $O(k)$ if a given hash function is injective from B_{8k} to $[(8k)^2]$. Therefore, in time $O(k\log(1/\epsilon)+k^2)$ and space $O(k^2)$, step 2 of the algorithm will construct the desired hash function H with a probability at least $1-\epsilon$. This is the only place in the algorithm where randomization is used.

With the hash function H constructed in step 2, we construct an array $B[1..(8k)^2]$ such that for each vertex v in B_{8k}, the array element $B[H(v)]$ keeps the vertex v as well as its e_{8k}-value. Now for any vertex w in the graph G, we can test in constant time if w is a vertex in the set B_{8k}, and in case it is, what is its e_{8k}-value.

Recall that we have constructed the set L_v^{8k} for each vertex v in B_{8k} in step 1. To construct the set E_R in step 3, we need to identify the edges in these sets that are in the trimmed subgraph G_T. Let $e=[v,w]$ be an edge in the set L_v^{8k} for a vertex v in B_{8k}. If w is not in B_{8k}, then since the e_{8k}-value of w is smaller than that of v, the edge e must be among the $8k$ heaviest edges incident to w. Thus, the edge e must be in the graph G_T. On the other hand, if w is in B_{8k}, then the edge e is in G_T if and only if $wt'(e)$ is not smaller than the e_{8k}-value of w. Thus, using the array $B[1..(8k)^2]$, we can test if the edge e is in the trimmed subgraph G_T in constant time. Finally, note that for an edge $e=[v,w]$ in L_v^{8k} where $v \in B_{8k}$, if w is not in B_{8k}, then the edge e appears in the set L_v^{8k} for exactly one vertex v in B_{8k}, while if w is in B_{8k}, then the edge e appears in both L_v^{8k} and L_w^{8k}. Therefore, for an edge $e=[v,w]$ with both v and w in B_{8k}, if we only consider the case when $v<w$, then we can avoid including multiple copies of an edge in the set E_R. Also note that the size of the set E_R is bounded by that of $\bigcup_{v\in B_{8k}} L_v^{8k}$, which is $O(k^2)$. In conclusion, the set E_R in step 3 can be constructed in time $O(k^2)$ and space $O(k^2)$.

Steps 4-5 add new edges in the trimmed subgraph G_T to the set E_R, and update the set E_R so that the set E_R only contains the $k(16k-1)$ heaviest edges seen so far. In order to keep the total processing time of steps 4-5 to $O(N)$, we, instead of adding a new vertex directly to the set E_R, use a buffer of size k^2 to keep the new edges found in steps 4-5. Only after we collect k^2 new edges in the buffer, we combine these k^2 new edges with those in the set E_R, and select the $k(16k-1)$ heaviest to form the new set E_R. By Lemma 4.1, this can be done in time $O(k^2)$ and space $O(k^2)$, contributing, in average, only constant time to each new edge. Also, to avoid including duplicated copies of an edge in the set E_R, for each edge $e=[v,w]$ encountered in steps 4-5 with $v \notin B_{8k}$ and $w \notin B_{8k}$, we only consider the edge when $v<w$. Putting all these together, we conclude that the total processing time of steps 4-5 is bounded by $O(N)$. The space complexity is $O(k^2)$.

Summarizing the above discussions proves the lemma. □

Now we return back to the p-WGM problem. Maximum matching on weighted graphs has been an extensively studied topic in theoretical computer science [30]. Currently, the best algorithm runs in time $O(n(m+n\log n))$ and space $O(m)$ on a weighted graph of n vertices and m edges [19,20], from which we can derive the following result.

Theorem 4.6. *There is an $O(k(m + n\log n))$-time and $O(m)$-space algorithm that on a weighted graph G of n vertices and m edges, either constructs a maximum k-matching in G or reports that no k-matching exists in G.*

Proof. This result is actually implied in the development of the $O(n(m + n\log n))$-time and $O(m)$-space algorithm due to Gabow [19,20] that constructs a maximum matching in a weighted graph. In the following, we provide the necessary proofs for the parts that are not explicitly given in [19,20] but are needed to achieve the stated result.

Let G be a weighted graph. For a set S of edges in G, we denote by $wt(S)$ the weight sum of the edges in S, and by $|S|$ the number of edges in S. Let M be a matching in the graph G. Again we define an *augmenting path* relative to M to be a simple path whose two ends are not covered by M and whose edges go alternatively between edges not in M and edges in M. The *weight-gain* of an augment path P relative to the matching M is defined to be $wt(P \setminus M) - wt(P \cap M)$. A *maximum augmenting path* relative to the matching M is an augmenting path whose weight-gain is the largest over all augmenting paths relative to M. For a weighted graph G, we have the following (recall that for two sets S_1 and S_2, $S_1 \oplus S_2 = (S_1 \setminus S_2) \cup (S_2 \setminus S_1)$):

Claim. *Let M_k be a maximum k-matching in the graph G, and let P be a maximum augmenting path relative to M_k, then $P \oplus M_k$ is a maximum $(k+1)$-matching in the graph G.*

For a proof of the claim, let M_{k+1} be a maximum $(k+1)$-matching in the graph G. Then all connected components C_1, C_2, …, C_h of the graph $M_k \oplus M_{k+1}$ are either a simple cycle or a simple path. Since $|M_{k+1}| = |M_k| + 1$, at least one of the components of $M_k \oplus M_{k+1}$ is an augmenting path relative to M_k. Without loss of generality, assume that the component C_h is an augmenting path relative to M_k, and let $C = C_1 \cup \cdots \cup C_{h-1}$. Then we have $|C \cap M_k| = |C \cap M_{k+1}|$. We claim that $wt(C \cap M_k) = wt(C \cap M_{k+1})$. In fact, if $wt(C \cap M_k) > wt(C \cap M_{k+1})$, then replacing the edges of the set $C \cap M_{k+1}$ in the $(k+1)$-matching M_{k+1} with the edges of the set $C \cap M_k$ would give a $(k+1)$-matching whose weight is larger than that of M_{k+1}, contradicting the assumption that M_{k+1} is a maximum $(k+1)$-matching. Similarly, if $wt(C \cap M_k) < wt(C \cap M_{k+1})$, then replacing the edges of the set $C \cap M_k$ in the k-matching M_k with the edges of the set $C \cap M_{k+1}$ would give a k-matching whose weight is larger than that of M_k, contradicting the assumption that M_k is a maximum k-matching. The equality $wt(C \cap M_k) = wt(C \cap M_{k+1})$ directly leads to the conclusion that the augmenting path C_h relative to M_k has its weight-gain equal to $wt(M_{k+1}) - wt(M_k)$. Since an augmenting path relative to M_k with a weight-gain larger than $wt(M_{k+1}) - wt(M_k)$ would give a $(k+1)$-matching whose weight is larger than that of the maximum $(k+1)$-matching M_{k+1}, we conclude that the path C_h is a maximum augmenting path relative to M_k, and augmenting the k-matching M_k with the maximum augmenting path C_h will result in the maximum $(k+1)$-matching M_{k+1}. This completes the proof of the claim.

The algorithm given by Gabow [19,20] is based on Edmonds' formulation of weighted matching as a linear program [11]. Starting with a maximum 0-matching M_0 (i.e., an empty set), for each $i = 0, 1\ldots$, the algorithm repeatedly finds a maximum augmenting path P_i relative to the maximum i-matching M_i, and augments the matching M_i along the path P_i to obtain a maximum $(i+1)$-matching M_{i+1} (whose correctness is given by the above claim). The process of finding a maximum augmenting path relative to a matching then augmenting the matching along the path is called a *phase*. Thus, after k phases, a maximum k-matching is constructed for the graph G. On the other hand, if the process is stopped for a maximum i-matching M_i with $i < k$ because there is no augmenting path relative to M_i, then we report that no k-matching exists in the graph G. Gabow [19,20] has developed an algorithm that implements the computation of a phase in the above process in time $O(m + n\log n)$ and space $O(m)$. Combining these two results gives the proof of the theorem. □

For an instance (G, k) of the p-WGM problem, the reduced subgraph G_R of the graph G contains $O(k^2)$ edges, thus no more than $O(k^2)$ vertices. Therefore, applying Theorem 4.6 to the reduced subgraph G_R, we conclude that a maximum k-matching in the reduced subgraph G_R can be constructed in time $O(k(k^2 + k^2\log k)) = O(k^3\log k)$ and space $O(k^2)$. Bringing this result into Lemma 4.5 and letting $\epsilon = 1/k^{k^2}$ give the following theorem.

Theorem 4.7. *There is a randomized algorithm for the p-WGM problem such that on an input (G, k) where G is a weighted graph of size N, with probability $1 - 1/k^{k^2}$, and running time $O(N + k^3\log k)$ and space $O(k^2)$, the algorithm either constructs a maximum k-matching in G or reports that no k-matching exists in G.*

We may not expect a very significant improvement on the complexity bounds given in Theorem 4.7, based on the current status of maximum matching algorithms for weighted graphs. Indeed, if we measure the complexity of the algorithms in terms of the number n of vertices in the graph, then the best algorithm for constructing a maximum weighted matching in a weighted graph takes time $O(n^3)$ [18]. Since a graph has to have at least $2k$ vertices in order to contain a k-matching, the best we may expect for our reduction algorithm is to reduce the input graph into a reduced graph G'_R of at least $2k$ vertices. Now applying the algorithm in [18] to the reduced graph G'_R will take time at least $O(k^3)$, which would give an algorithm of time $O(N + k^3)$ for the p-WGM problem. We also remark that directly applying the algorithm of time $O(n^3)$ in [18] to the reduced subgraph G_R in Lemma 4.5 does not give a better bound: the reduced subgraph G_R in Lemma 4.5 may have $\Omega(k^2)$ vertices.

Again, there seem no known algorithms that are specifically for solving the p-WGM problem. Chitnis et al. [5] studied the p-WGM problem on the dynamic graph streaming model, and proposed two randomized algorithms. As our discussions on the algorithms in [5] for the p-UGM problem (see Section 3), we may remove the intricate (and expensive) operations that deal with edge deletions in the algorithms given in [5], so that the algorithms can be used for solving the p-WGM problem. With this simplification, in order to have a success probability $1-\epsilon$, the first streaming algorithm proposed in [5] would have update time (i.e., the time between reading two consecutive elements in the input) at least $O(\log W \log(1/\epsilon))$ and use space $O(k^4 W \log(1/\epsilon))$, where W is the number of different values in the edge weights. As a consequence, if we use this algorithm to solve the p-WGM problem, the algorithm runs in time at least $O(\log W \log(1/\epsilon)N + k^4 W \log(1/\epsilon))$ and uses space $O(k^4 W \log(1/\epsilon))$. If we use the second algorithm proposed in [5], with the above simplification, to solve the p-WGM problem, we would get an algorithm with running time at least $O(N \log k \log W + k^2 W \log(1/\epsilon))$ and space $O(k^2 W \log(1/\epsilon))$. More seriously, the second algorithm requires that the input weighted graphs have no matching of size larger than k, which makes the algorithm to be applicable to a much restricted class of graphs. In comparison, our algorithm in Theorem 4.7 solves the p-WGM problem in time $O(N + k^3 \log k)$ and space $O(k^2)$ with a very high success probability $1 - 1/k^{k^2}$, and with no constraint on the structures of the input graph.

5. Conclusion and final remarks

Motivated by the recent algorithmic research in massive data processing, we proposed a multivariate computational model whose complexity bounds are measured by both input size N and a parameter k, where N is supposed to be extremely large while the parameter k is a measure for the power of local resources (i.e., computational time and space) that can be used to deal with the massive data. We have used classical problems in computational optimization, the graph matching problems on both unweighted and weighted graphs, as examples to show how our model is used in effectively dealing with classical computational problems in massive data processing. In particular, we show how we can spend a linear-time pre-processing on the massive input data, with limited local memory space, to reduce a problem instance to an instance that is manageable by the limited local resources. Moreover, we showed how the local resources can be effectively managed to achieve the best or nearly best possible usage. In particular, we have presented an algorithm that finds a k-matching in an unweighted graph of size N in time $O(N + k^{2.5})$ and space $O(k^2)$, and an algorithm that constructs a maximum weighted k-matching in a weighted graph of size N in time $O(N + k^3 \log k)$ and space $O(k^2)$.

Our algorithms for the graph matching problems are randomized algorithms, with exponentially small error bounds. If we use a balanced search tree to support the search and insertion operations in our process of large-vertices, instead of using injective hash functions, then our randomized algorithms will become deterministic algorithms. However, in their deterministic versions, our algorithm for solving the p-UGM problem in Theorem 3.6 will run in time $O(N \log k + k^{2.5})$ and space $O(k^2)$, and our algorithm for solving the p-WGM problem in Theorem 4.7 will run in time $O(N \log k + k^3 \log k)$ and space $O(k^2)$.

The computational model we studied in the current paper suggests reconsideration for many computational problems, including many classical ones, in the framework of massive data processing where the inputs are supposed to have extremely large size. For example, for two given vertices s and t in a weighted graph of size N, can we construct an st-path of length bounded by k whose weight is the minimum over all st-paths of length bounded by k in time $O(N + f_1(k))$ and space $O(f_2(k))$, where $f_1(k)$ and $f_2(k)$ are functions of the parameter k? If the answer is yes, what is the best we can get for $f_1(k)$ and $f_2(k)$? Note that Thorup's linear-time algorithm [31] for the single-source shortest path problem seems not directly applicable here because of the constraints on space complexity.

A particular research area where our model can be investigated is kernelization algorithms in parameterized computation [17]. Instances of a parameterized problem Q take the format (x, k), where k is the parameter. A *kernelization algorithm* for the problem Q on an input (x, k) produces an instance (x', k') such that (x, k) is a yes-instance of Q if and only if (x', k') is a yes-instance of Q, and that the size of x' and the value of the new parameter k' are both bounded by a function of the original parameter k that is independent of the size of the original input (x, k). Most proposed kernelization algorithms run in polynomial time and were developed without much consideration on the efficiency of the algorithms. Recently, there have been studies on linear-time kernelization algorithms [25]. On the other hand, space complexity has rarely been considered in kernelization algorithms. Many kernelization algorithms, including those proposed in [25], are based on the techniques that remove or modify "obvious" structures in the input, which, intrinsically, require space for storing the input and recording the changes, leading to demand of a large amount of space, and in many cases also to demand of super-linear time. On the other hand, the approach of kernelization seems to fit very well in dealing with massive data, and provides reduction and pre-processing techniques to reduce problem instances of very large size to instances of much small (thus manageable) size. In particular, kernelization algorithms whose running time is linear or nearly linear in terms of the input size, with limited space, are very interesting in this direction of research. We have initialized this line of research and obtained some interesting results [2].

There seem to be some very interesting relations between our proposed model and the streaming algorithm model. A streaming algorithm S can be converted into an algorithm on our model, which can keep the same space complexity with the running time equal to the update time of S times the input size N. On the other hand, some techniques developed for our model may become useful for developing streaming algorithms. For example, some techniques presented in the current paper have been used in the development of improved streaming algorithms for graph matching [3]. However, there are

also essential differences between the two models. For example, it has been proved that the space complexity for streaming algorithms on the dynamic streaming model for maximum k-matching is $\Omega(Wk^2)$, where W is the number of different edge weights in the input graph. On the other hand, the algorithm presented in the current paper for maximum k-matching in weighted graphs has space complexity $O(k^2)$, which is independent of the value W. Further study of the relationship between the two models should be interesting.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] J. Alman, M. Mnich, V.V. Williams, Dynamic parameterized problems and algorithms, in: Proc. 44th International Colloquium on Automata, Languages and Programming (ICALP 2017), 2017, 41.
[2] J. Chen, Q. Huang, I.A. Kanj, G. Xia, Near-optimal algorithms for point-line covering problems, in: Proc. 39th International Symp. on Theoretical Aspects of Computer Science (STACS 2022), 2022, 21.
[3] J. Chen, Q. Huang, I.A. Kanj, Q. Li, G. Xia, Streaming algorithms for graph k-matching with optimal or near-optimal update time, in: Proc. 32nd International Symp. on Algorithms and Computation (ISAAC 2021), 2021, 48.
[4] R. Chitnis, G. Cormode, Towards a theory of parameterized streaming algorithms, in: Proc. 14th International Symp. on Parameterized and Exact Computation (IPEC 2019), 2019, 7.
[5] R. Chitnis, G. Cormode, H. Esfandiari, M. Hajiaghayi, A. McGregor, M. Monemizadeh, Kernelization via sampling with applications to finding matchings and related problems in dynamic graph streams, in: Proc. 27th ACM-SIAM Symp. on Discrete Algorithms (SODA 2016), 2016, pp. 1326–1344.
[6] R. Chitnis, G. Cormode, H. Esfandiari, M. Hajiaghayi, M. Monemizadeh, New streaming algorithms for parameterized maximal matching and beyond, in: Proc. 27th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA 2015), 2015, pp. 56–58.
[7] R. Chitnis, G. Cormode, M. Hajiaghayi, M. Monemizadeh, Parameterized streaming: maximal matching and vertex cover, in: Proc. 26th ACM-SIAM Symp. on Discrete Algorithms (SODA 2015), 2015, pp. 1234–1251.
[8] S.A. Cook, Deterministic CFL's are accepted simultaneously in polynomial time and log squared space, in: Proc. 11th ACM Symp. on Theory of Computing (STOC 79), 1979, pp. 338–345.
[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 2nd ed., The MIT Press, Cambridge, MA, 2009.
[10] G. Cormode, D. Firmani, A unifying framework for l_0-sampling algorithms, Distrib. Parallel Databases 32 (3) (2014) 315–335.
[11] J. Edmonds, Maximum matching and a polyhedron with 0, 1-vertices, J. Res. Natl. Bur. Stand. 69B (1965) 125–130.
[12] H. Esfandiari, M. Hajiaghayi, V. Liaghat, M. Monemizadeh, K. Onak, Streaming algorithms for estimating the matching size in planar graphs and beyond, ACM Trans. Algorithms 14 (4) (2018) 1–23.
[13] S. Fafianie, S. Kratsch, Streaming kernelization, in: Proc. 39th International Symp. on Math. Foundations of Computer Science (MFCS 2014), 2014, pp. 275–286.
[14] W. Fan, F. Geerts, F. Neven, Making queries tractable on big data with proprecessing, in: Proc. 39th International Conference on Very Large Data Bases (VLDB 2013), 2013, pp. 685–696.
[15] W. Fan, C. Hu, Big graph analysis: from queries to dependencies and association rules, Data Sci. Eng. 2 (1) (2017) 36–55.
[16] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, J. Zhang, On graph problems in a semi-streaming model, Theor. Comput. Sci. 348 (2–3) (2005) 207–216.
[17] F. Fomin, D. Lokshtanov, S. Saurabh, M. Zehavi, Kernelization: Theory of Parameterized Preprocessing, Cambridge University Press, 2019.
[18] H.N. Gabow, Implementations of Algorithms for maximum Matching on Nonbipartite Graphs, Ph.D. Dissertation, Comp. Sci. Dept., Stanford University, CA, 1973.
[19] H.N. Gabow, Data structures for weighted matching and nearest common ancestors with linking, in: Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1990), 1990, pp. 434–443.
[20] H.N. Gabow, Data structures for weighted matching and extensions by b-matching and f-factors, ACM Trans. Algorithms 14 (3) (2018) 39.
[21] I. Grujic, S. Bogdanovic-Dinic, L. Stoimenov, Collecting and analyzing data from e-government Facebook pages, in: ICT Innovations, 2014.
[22] J.E. Hopcroft, R.M. Karp, An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs, SIAM J. Comput. 2 (4) (1973) 225–231.
[23] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: Proc. 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10), 2010, pp. 135–145.
[24] A. McGregor, Graph stream algorithms: a survey, SIGMOD Rec. 43 (1) (2014) 9–20.
[25] G.B. Mertzios, A. Nichterlein, R. Niedermeier, The power of linear-time data reduction for maximum matching, in: Proc. 42nd International Symp. on Math. Foundations of Computer Science (MFCS 2017), 2017, 46.
[26] S. Micali, V.V. Vazirani, An $O(\sqrt{V}E)$ algorithm for finding maximum matching in general graphs, in: Proc. 21st IEEE Symp. on Foundations of Computer Science (FOCS, vol. 80, 1980, pp. 17–27.
[27] M. Mucha, P. Sankowski, Maximum matchings via Gaussian elimination, in: Proc. 45th IEEE Symp. on Foundations of Computer Science (FOCS 2004), 2004, pp. 248–255.
[28] S. Muthukrishna, Data streams: algorithms and applications, Found. Trends Theor. Comput. Sci. 1 (2) (2005) 117–236.
[29] R. Rubinfeld, A. Shapira, Sublinear time algorithms, SIAM J. Discrete Math. 25 (4) (2011) 1562–1588.
[30] A. Schrijver, Combinatorial Optimization: Polyhedra and Efficiency, Algorithms and Combinatorics, Springer-Verlag, Berlin, Heidelberg, 2003.
[31] M. Thorup, Undirected single-source shortest paths with positive integer weights in linear time, J. ACM 46 (3) (1999) 362–394.
[32] V.V. Vazirani, A theory of alternating paths and blossoms for proving correctness of the $O(\sqrt{V}E)$ general graph maximum matching algorithm, Combinatorica 14 (1) (1994) 71–109.
[33] V.V. Vazirani, A simplification of the MV matching algorithm and its proof, arXiv:1210.4594v5 [cs.DS], 2013.