An Algorithmic Study of Fully Dynamic Independent Sets for Map Labeling

Sujoy Bhore
Algorithms and Complexity Group, TU Wien, Austria
sujoy@ac.tuwien.ac.at

Guangping Li
Algorithms and Complexity Group, TU Wien, Austria
guangping@ac.tuwien.ac.at

Martin Nöllenburg
Algorithms and Complexity Group, TU Wien, Austria
noellenburg@ac.tuwien.ac.at

Abstract

Map labeling is a classical problem in cartography and geographic information systems (GIS) that asks to place labels for area, line, and point features, with the goal to select and place the maximum number of independent, i.e., overlap-free, labels. A practically interesting case is point labeling with axis-parallel rectangular labels of common size. In a fully dynamic setting, at each time step, either a new label appears or an existing label disappears. Then, the challenge is to maintain a maximum cardinality subset of pairwise independent labels with sub-linear update time. Motivated by this, we study the maximal independent set (MIS) and maximum independent set (Max-IS) problems on fully dynamic (insertion/deletion model) sets of axis-parallel rectangles of two types – (i) uniform height and width and (ii) uniform height and arbitrary width; both settings can be modeled as rectangle intersection graphs.

We present the first deterministic algorithm for maintaining a MIS (and thus a 4-approximate Max-IS) of a dynamic set of uniform rectangles with amortized sub-logarithmic update time. This breaks the natural barrier of $\Omega(\Delta)$ update time (where Δ is the maximum degree in the graph) for *vertex updates* presented by Assadi et al. (STOC 2018). We continue by investigating Max-IS and provide a series of deterministic dynamic approximation schemes. For uniform rectangles, we first give an algorithm that maintains a 4-approximate Max-IS with $O(1)$ update time. In a subsequent algorithm, we establish the trade-off between approximation quality $2(1 + \frac{1}{k})$ and update time $O(k^2 \log n)$, for $k \in \mathbb{N}$. We conclude with an algorithm that maintains a 2-approximate Max-IS for dynamic sets of unit-height and arbitrary-width rectangles with $O(\omega \log n)$ update time, where ω is the maximum size of an independent set of rectangles stabbed by any horizontal line. We have implemented our algorithms and report the results of an experimental comparison exploring the trade-off between solution quality and update time for synthetic and real-world map labeling instances.

2012 ACM Subject Classification Theory of computation → Computational geometry; Theory of computation → Dynamic graph algorithms

Keywords and phrases Independent Sets, Dynamic Algorithms, Rectangle Intersection Graphs, Approximation Algorithms, Experimental Evaluation

Digital Object Identifier 10.4230/LIPIcs.ESA.2020.19

Related Version A full version of the paper is available at `https://arxiv.org/abs/2002.07611`.

Supplementary Material Source code and benchmark data at `https://dyna-mis.github.io/dynaMIS/`.

Funding Research supported by the Austrian Science Fund (FWF), grant P 31119.

1 Introduction

Map Labeling is a classical problem in cartography and geographic information systems (GIS), that has received significant attention in the past few decades and is concerned with selecting and positioning labels on a map for area, line, and point features. The focus in the computational geometry community has been on labeling point features [3, 20, 39, 40]. The labels are typically modeled as the bounding boxes of short names, which correspond precisely to unit height, but arbitrary width rectangles; alternatively, labels can be standardized icons or symbols, which correspond to rectangles of uniform size. In map labeling, a key task is in fact to select an independent (i.e., overlap-free) set of labels from a given set of candidate labels. Commonly the optimization goal is related to maximizing the number of labels. Given a set $\mathcal{R}$ of rectangular labels, Map Labeling is essentially equivalent to the problem of finding a maximum independent set in the intersection graph induced by $\mathcal{R}$.

The independent set problem is a fundamental graph problem with a wide range of applications. Given a graph $G = (V, E)$, a set of vertices $M \subset V$ is *independent* if no two vertices in M are adjacent in G. A *maximal independent set* (MIS) is an independent set that is not a proper subset of any other independent set. A *maximum independent set* (Max-IS) is a maximum cardinality independent set. While Max-IS is one of Karp's 21 classic NP-complete problems [31], computing a MIS can easily be done by a simple greedy algorithm in $O(|E|)$ time. The MIS problem has been studied in the context of several other prominent problems, e.g., graph coloring [33], maximum matching [30], and vertex cover [36]. On the other hand, Max-IS serves as a natural model for many real-life optimization problems, including map labeling [3], computer vision [6], information retrieval [37], and scheduling [38].

Stronger results for independent set problems in geometric intersection graphs are known in comparison to general graphs. For instance, it is known that Max-IS on general graphs cannot be approximated better than $|V|^{1-\epsilon}$ in polynomial time for any $\epsilon > 0$ unless NP=ZPP [27]. In contrast, a randomized polynomial-time algorithm exists that computes for rectangle intersection graphs an $O(\log \log n)$-approximate solution to Max-IS with high probability [12], as well as QPTASs [2,16]. The Max-IS problem is already NP-Hard on unit square intersection graphs [21], however, it admits a polynomial-time approximation scheme (PTAS) for unit square intersection graphs [19] and more generally for pseudo disks [13]. Moreover, for rectangles with either uniform size or at least uniform height and bounded aspect ratio, the size of an MIS is not arbitrarily worse than the size of a Max-IS. For instance, any MIS of a set of uniform rectangles is a 4-approximate solution to the Max-IS problem, since each rectangle can have at most four independent neighbors.

Past research has mostly considered static label sets in static maps [3, 20, 39, 40] and in dynamic maps allowing zooming [7] or rotations [25], but not fully dynamic label sets with insertions and deletions of labels. Recently, Klute et al. [32] proposed a framework for semi-automatic label placement, where domain experts can interactively insert and delete labels. In their setting an initially computed large independent set of labels can be interactively modified by a cartographer, who can easily take context information and soft criteria such as interactions with the background map or surrounding labels into account. Standard map labeling algorithms typically do not handle such aspects well. Based on these modifications (such as deletion, forced selection, translation, or resizing), the solution is updated by a dynamic algorithm while adhering to the new constraints. Another scenario for dynamic labels are maps, in which features and labels (dis-)appear over time, e.g., based on a stream of geotagged, uniform-size photos posted on social media or, more generally, maps with labels of dynamic spatio-temporal point sets [22]. For instance, a geo-located event

that happens at time t triggers the availability of a new label for a certain period of time, after which it vanishes again. Examples beyond social media are reports of earth quakes, forest fires, or disease incidences. Motivated by this, we study the independent set problem for dynamic rectangles of two types – (i) uniform height and width and (ii) uniform height and arbitrary width. We consider fully dynamic algorithms for maintaining independent sets under insertions and deletions of rectangles, i.e., vertex insertions and deletions in the corresponding dynamic rectangle intersection graph.

Dynamic graphs are subject to discrete changes over time, i.e., insertions or deletions of vertices or edges [18]. A dynamic graph algorithm solves a computational problem, such as the independent set problem, on a dynamic graph by updating efficiently the previous solution as the graph changes over time, rather than recomputing it from scratch. A dynamic graph algorithm is called *fully dynamic* if it allows both insertions and deletions, and *partially dynamic* if only insertions or only deletions are allowed. While general dynamic independent set algorithms can obviously also be applied to rectangle intersection graphs, our goal is to exploit their geometric properties to obtain more efficient algorithms.

Related Work. There has been a lot of work on dynamic graph algorithms in the last decade and dynamic algorithms still receive considerable attention in theoretical computer science. We point out some of these works, e.g., on spanners [9], vertex cover [10], set cover [1], graph coloring [11], and maximal matching [23]. In particular, the maximal independent set problem on dynamic graphs with edge updates has attracted significant attention in the last two years [4, 5, 8, 15, 17]. Recently, Henzinger et al. [28] studied the Max-IS problem for intervals, hypercubes and hyperrectangles in d dimensions, with special assumptions. They assumed that the objects are axis-parallel and contained in the space $[0, N]^d$; the value of N is given in advance, and each edge of an input object has length at least 1 and at most N. Moreover, they have designed dynamic approximation algorithms and lower bounds, where the update time depends on N and is of high complexity. Gavruskin et al. [24] studied the Max-IS problem for dynamic proper intervals (intervals cannot contain one another), and showed how to maintain a Max-IS with polylogarithmic update time.

Results and Organization. We study MIS and Max-IS problems for dynamic sets of $O(n)$ axis-parallel rectangles of two types: (i) congruent rectangles of uniform height and width and (ii) rectangles of uniform height and arbitrary width. For both classes of rectangles a MIS can be maintained in $\Omega(\Delta)$ update time by using the recent algorithm of Assadi et al. [4], where Δ is the maximum degree of the intersection graph. A $(1+\epsilon)$-approximate Max-IS can be maintained for unit squares in $O(n^{1/\epsilon^2})$ time [19], and a $(1+\frac{1}{k})$-approximate Max-IS can be maintained for unit height and arbitrary width rectangles in $O(n^{2k-1})$ update time [3] for any integer $k \geq 1$. In this paper we design and implement algorithms for dynamic MIS and Max-IS that demonstrate the trade-off between update time and approximation factor, both from a theoretical perspective and in an experimental evaluation. In contrast to the recent dynamic MIS algorithms, which are randomized [4, 5, 8, 15], our algorithms are deterministic.

In Section 3 we present an algorithm that maintains a MIS of a dynamic set of unit squares in amortized $O(\log^{2/3+o(1)} n)$ update time, improving the best known update time $\Omega(\Delta)$ [4]. A major, but generally unavoidable bottleneck of that algorithm is that the entire graph is stored explicitly, and thus insertions/deletions of vertices take $\Omega(\Delta)$ time. We use structural geometric properties of the unit squares along with a dynamic orthogonal range searching data structure to bypass the explicit intersection graph and break this bottleneck.

In Section 4, we study the Max-IS problem. For dynamic unit squares, we give an algorithm that maintains a 4-approximate Max-IS with $O(1)$ update time. We generalize this algorithm and improve the approximation factor to $2(1+\frac{1}{k})$, which increases the update time to $O(k^2 \log n)$. We conclude with an algorithm that maintains a 2-approximate Max-IS for a dynamic set of unit-height and arbitrary-width rectangles (in fact, for a dynamic interval graph, which is of independent interest) with $O(\omega \log n)$ output-sensitive update time, where ω is the maximum size of an independent set of rectangles stabbed by any horizontal line.

Finally, Section 5 provides an experimental evaluation of the proposed Max-IS approximation algorithms on synthetic and real-world map labeling data sets using unit squares. The experiments explore the trade-off between solution size and update time, as well as the speed-up of the dynamic algorithms over their static counterparts. See the supplemental material for source code and benchmark data.

Proofs marked ($\star$) *are missing due to space constraints; refer to the full version for all details.*

2 Model and Notation

Let $\mathcal{R} = \{r_1, \ldots, r_\nu\}$ be a set of ν axis-parallel, unit-height rectangles in the plane. If the rectangles are of uniform height and width, we can use an affine transformation to map $\mathcal{R}$ to a set of unit squares $\mathcal{S} = \{s_1, \ldots, s_\nu\}$ instead. We use the shorthand notation $[n] = \{1, 2, \ldots, n\}$. In our setting we assume that $\mathcal{R}$ is dynamically updated by a sequence of $N \in \mathbb{N}$ insertions and deletions. We denote the set of rectangles at step $i \in [N]$ as $\mathcal{R}_i$. For a set of unit squares $\mathcal{S}_i = \{s_1, \ldots, s_\nu\}$ at step $i \in [N]$ we further define the set $\mathcal{C}_i = \{c_1, \ldots, c_\nu\}$ of the corresponding square centers. Let $n = \max\{|\mathcal{R}_i| \mid i \in [N]\}$ be the maximum number of rectangles over all steps. The rectangle intersection graph defined by $\mathcal{R}_i$ at time step i is denoted as $G_i = (\mathcal{R}_i, E_i)$, where two rectangles $r, r' \in \mathcal{R}_i$ are connected by an edge $\{r, r'\} \in E_i$ if and only if $r \cap r' \neq \emptyset$. We use M_i to denote a maximal independent set in G_i, and OPT_i to denote a maximum independent set in G_i. For a graph $G = (V, E)$ and a vertex $v \in V$, let $N(v)$ denote the set of neighbors of v in G. This notation also extends to any subset $U \subseteq V$ by defining $N(U) = \bigcup_{v \in U} N(v)$. We use $\deg(v)$ to denote the degree of a vertex $v \in V$. For any vertex $v \in V$, let $N^r(v)$ be the r-neighborhood of v, i.e., the set of vertices that are within distance at most r from v (excluding v).

3 Dynamic MIS with Sub-Logarithmic Update Time

In this section, we study the MIS problem for dynamic uniform rectangles. As stated before we can assume w.l.o.g. that the rectangles are unit squares. We design an algorithm that maintains a MIS for a dynamic set of $O(n)$ unit squares in sub-logarithmic update time. Assadi et al. [4] presented an algorithm for maintaining a MIS on general dynamic graphs with $\Omega(\Delta)$ update time, where Δ is the maximum degree in the graph. In the worst case, however, that algorithm takes $O(n)$ update time. In fact, it seems unavoidable for an algorithm that explicitly maintains the (intersection) graph to perform a MIS update in less than $\Omega(\deg(v))$ time for an insertion/deletion of a vertex v. In contrast, our proposed algorithm in this section does not explicitly maintain the intersection graph $G_i = (\mathcal{S}_i, E_i)$ (for any $i \in [N]$), but rather only the set of squares $\mathcal{S}_i$ in a suitable dynamic geometric data structure. For the ease of explanation, however, we do use graph terms at times.

Let $i \in [N]$ be any time point in the sequence of updates. For each square $s_v \in \mathcal{S}_i$, let s_v^a be a square of side length a concentric with s_v. Further, let M_i denote the MIS that we compute for $G_i = (\mathcal{S}_i, E_i)$, and let $\mathcal{C}(M_i) \subseteq \mathcal{C}_i$ be their corresponding square centers. We

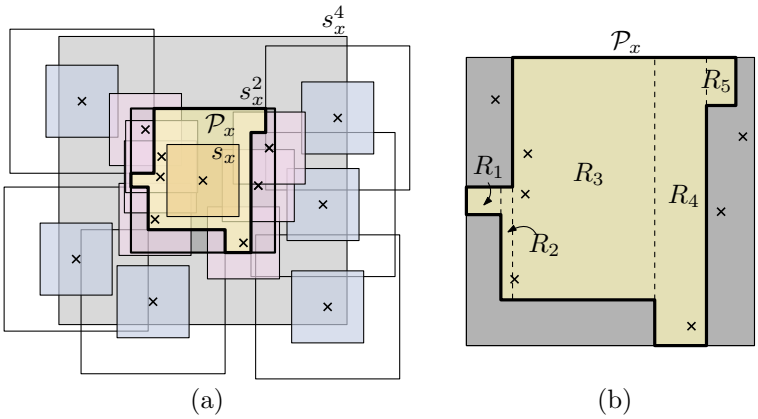


Figure 1 Example for the deletion of a square s_x. (a) Square s_x, its neighborhood with centers in s_x^2, its 2-neighborhood with centers in s_x^4, and the polygon $\mathcal{P}_x$. (b) Vertical slab partition of $\mathcal{P}_x$.

maintain two fully dynamic orthogonal range searching data structures throughout: (i) a dynamic range tree $T(\mathcal{C}_i)$ for the entire point set $\mathcal{C}_i$ and (ii) a dynamic range tree $T(\mathcal{C}(M_i))$ for the point set $\mathcal{C}(M_i)$ corresponding to the centers of M_i. They can be implemented with dynamic fractional cascading [34], which yields $O(\log n \log\log n)$ update time and $O(k + \log n \log\log n)$ query time for reporting k points. The currently best fully dynamic data structure for orthogonal range reporting requires $O(\log^{2/3+o(1)} n)$ amortized update time and $O(k + \frac{\log n}{\log\log n})$ amortized query time [14].

We compute the initial MIS M_1 for $G_1 = (\mathcal{S}_1, E_1)$ by using a simple linear-time greedy algorithm. First we initialize the range tree $T(\mathcal{C}_i)$. Then we iterate through the set $\mathcal{S}_1$ as long as it is not empty, select a square s_v for M_1 and insert its center into $T(\mathcal{C}(M_i))$, find its neighbors $N(s_v)$ by a range query in $T(\mathcal{C}_i)$ with the concentric square s_v^2, and delete $N(s_v)$ from $\mathcal{S}_1$. It is clear that once this process terminates, M_1 is a MIS.

When we move in the next step from $G_i = (\mathcal{S}_i, E_i)$ to $G_{i+1} = (\mathcal{S}_{i+1}, E_{i+1})$, either a square is inserted into $\mathcal{S}_i$ or deleted from $\mathcal{S}_i$. Let s_x be the square that is inserted or deleted.
Insertion: When we insert a square s_x into $\mathcal{S}_i$ to obtain $\mathcal{S}_{i+1}$, we do the following operations. First, we obtain $T(\mathcal{C}_{i+1})$ by inserting the center of s_x into $T(\mathcal{C}_i)$. Next, we have to detect whether s_x can be included in M_{i+1}. If there exists a square s_u from M_i intersecting s_x, we should not include s_x; otherwise we will add it to the MIS. To check this, we search with the range s_x^2 in $T(\mathcal{C}(M_i))$. By a simple packing argument, we know that no more than four points (the centers of four independent squares) of $\mathcal{C}(M_i)$ can be in the range s_x^2. If the query returns such a point, then s_x would intersect with another square in M_i and we set $M_{i+1} = M_i$. Otherwise, we insert s_x into $T(\mathcal{C}(M_i))$ to get $T(\mathcal{C}(M_{i+1}))$.
Deletion: When we delete a square s_x from $\mathcal{S}_i$, it is possible that $s_x \in M_i$. In this case we may have to add squares from $N(s_x)$ into M_{i+1} to keep it maximal. Since any square can have at most four independent neighbors, we can add in this step up to four squares to M_{i+1}.

First, we check if $s_x \in M_i$. If not, then we simply delete s_x from $T(\mathcal{C}_i)$ to get $T(\mathcal{C}_{i+1})$ and set $M_{i+1} = M_i$. Otherwise, we delete again s_x from $T(\mathcal{C}_i)$ and also from $T(\mathcal{C}(M_i))$. In order to detect which neighbors of s_x can be added to M_i, we use suitable queries in the data structures $T(\mathcal{C}(M_i))$ and $T(\mathcal{C}_i)$. Figure 1a illustrates the next observations. The centers of all neighbors in $N(s_x)$ must be contained in the square s_x^2. But some of these neighbors may intersect other squares in M_i. In fact, these squares would by definition belong to the 2-neighborhood, i.e., be in the set $Q_x = N^2(s_x) \cap M_i$. We can obtain Q_x by querying

$T(\mathcal{C}(M_i))$ with the range s_x^4. Since $s_x \in M_i$, we know that $Q_x \cap s_x^2 = \emptyset$ and hence the center points of the squares in Q_x lie in the annulus $s_x^4 - s_x^2$. A simple packing argument implies that $|Q_x| \leq 12$ and therefore querying $T(\mathcal{C}(M_i))$ will return at most 12 points.

Next we define the rectilinear polygon $\mathcal{P}_x = s_x^2 - \bigcup_{s_y \in Q_x} s_y^2$, which contains all possible center points of squares that are neighbors of s_x but do not intersect any square $s_y \in M_i \setminus \{s_x\}$.

▶ **Observation 1** (⋆). *The polygon $\mathcal{P}_x$ has at most 28 corners.*

Next we want to query $T(\mathcal{C}_i)$ with the range $\mathcal{P}_x$, which we do by vertically partitioning $\mathcal{P}_x$ into rectangular slabs $R_1, \ldots, R_c$ for some $c \leq 28$ (see Figure 1b). For each slab R_j, where $1 \leq j \leq c$, we perform a range query in $T(\mathcal{C}_i)$. If a center p is returned, we can add the corresponding square s_p into M_{i+1}, and p into $T(\mathcal{C}(M_i))$ to obtain $T(\mathcal{C}(M_{i+1}))$. Moreover, we have to update $\mathcal{P}_x \leftarrow \mathcal{P}_x - s_p$, refine the slab partition and continue querying $T(\mathcal{C}_i)$ with the slabs of $\mathcal{P}_x$. We know that the deleted square s_x can have at most four independent neighbors. So after adding at most four new squares to M_{i+1} we know that $\mathcal{P}_x = \emptyset$ and we can stop searching.

▶ **Lemma 2** (⋆). *The set M_i is a maximal independent set of $G_i = (\mathcal{S}_i, E_i)$ for each step $i \in [N]$.*

Both the Insertion and the Deletion operations consist of (i) a constant number of insertions or deletions in the two fully dynamic orthogonal range searching data structures and (ii) of a constant number of orthogonal range reporting queries. These queries return at most 12 points in $T(\mathcal{C}(M_{i-1}))$. While the query range $\mathcal{P}_x$ for $T(\mathcal{C}_{i-1})$ in the Deletion operation may contain many points, an arbitrary point in $\mathcal{P}_x$ is sufficient for adding a new square to the independent set. Thus we do not need to enumerate all contained points, but just a single witness. In the orthogonal range searching data structure of Chan and Tsakalidis [14], the amortized update time for an insertion/deletion is $O(\log^{2/3+o(1)} n)$; this dominates the query time and together with Lemma 2 yields:

▶ **Theorem 3.** *We can maintain a maximal independent set of a dynamic set of unit squares, deterministically, in amortized $O(\log^{2/3+o(1)} n)$ update time.*

Proof. The correctness follows from Lemma 2. It remains to show the running time for the fully dynamic updates. At each step i we perform either an Insertion or a Deletion operation. Let us first discuss the update time for the insertion of a square. As described above, an insertion performs one or two insertions of the center of the square into the range trees and one range query in $T(\mathcal{C}(M_{i-1}))$, which will return at most four points. Using dynamic fractional cascading [34], this requires $O(\log n \log\log n)$ time; with the data structure of Chan and Tsakalidis [14], the amortized update time for inserting a square is $O(\log^{2/3+o(1)} n)$, the time for inserting a new point into their range searching data structure; this dominates the query time. The deletion of a square triggers either just a single deletion from the range tree $T(\mathcal{C}_{i-1})$ or, if it was contained in the MIS M_{i-1}, two deletions, up to four insertions, and a sequence of range queries: one query in $T(\mathcal{C}(M_{i-1}))$, which can return at most 12 points and a constant number of queries in $T(\mathcal{C}_{i-1})$ with the constant-complexity slab partition of $\mathcal{P}_x$. Note that while the number of points in $\mathcal{P}_x$ can be large, for our purpose it is sufficient to return a single point in each query range if it is not empty. Therefore, the update time for a deletion is again $O(\log n \log\log n)$ with dynamic fractional cascading [34] or amortized $O(\log^{2/3+o(1)} n)$ [14], depending on the selected data structure. ◀

For unit square intersection graphs, recall that any square in a MIS can have at most four mutually independent neighbors. Therefore, maintaining a dynamic MIS immediately implies maintaining a dynamic 4-approximate Max-IS.

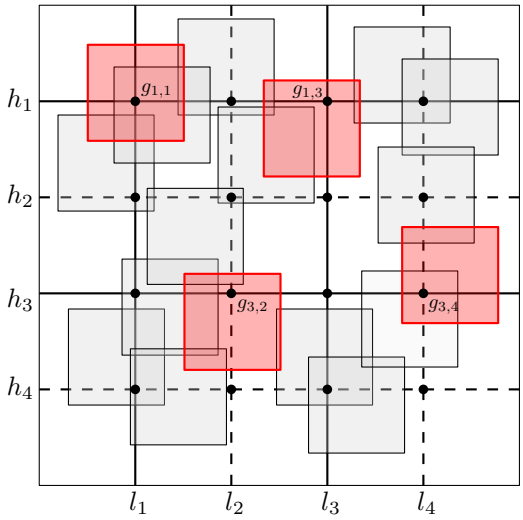


Figure 2 Example instance with bounding square $\mathcal{B}$ partitioned into a 5×5 grid. Red squares represent the computed 4-approximate solution, which here is $M(O(H))$.

▶ **Corollary 4.** *We can maintain a 4-approximate maximum independent set of a dynamic set of unit squares, deterministically, in amortized* $O(\log^{2/3+o(1)} n)$ *update time.*

4 Approximation Algorithms for Dynamic Maximum Independent Set

In this section, we study the Max-IS problem for dynamic unit squares as well as for unit-height and arbitrary-width rectangles. In a series of dynamic schemes proposed in this section, we establish the trade-off between the update time and the solution size, i.e., the approximation factors. First, we design a 4-approximation algorithm with $O(1)$ update time for Max-IS on dynamic unit squares (Section 4.1). We improve this to an algorithm that maintains a $2(1+\frac{1}{k})$-approximate Max-IS with $O(k^2 \log n)$ update time (Section 4.2). Finally, we conclude with an algorithm that deterministically maintains a 2-approximate Max-IS with output-sensitive $O(\omega \log n)$ update time, where ω is the maximum size of an independent set of the unit-height rectangles stabbed by any horizontal line (Section 4.3).

Let $\mathcal{B}$ be a bounding square of the dynamic set of 1×1-unit squares $\bigcup_{i\in[N]} \mathcal{S}_i$ of side length $\kappa \times \kappa$; we can assume that $\kappa = O(n)$; otherwise we could contract empty horizontal/vertical strips of $\mathcal{B}$. Let $H = \{h_1, \ldots, h_\kappa\}$ and $L = \{l_1, \ldots, l_\kappa\}$ be a set of top-to-bottom and left-to-right ordered equidistant horizontal and vertical lines partitioning $\mathcal{B}$ into a square grid of side-length-1 cells, see Figure 2. Let $E_H = \{h_i \in H \mid i = 0 \pmod 2\}$ and $O_H = \{h_i \in H \mid i = 1 \pmod 2\}$ be the set of even and odd horizontal lines, respectively.

4.1 4-Approximation Algorithm with Constant Update Time

We design a 4-approximation algorithm for the Max-IS problem on dynamic unit square intersection graphs with constant update time. Our algorithm is based on a grid partitioning approach. Consider the square grid on $\mathcal{B}$ induced by the sets H and L of horizontal and vertical lines. We denote the grid points as $g_{p,q}$ for $p, q \in [\kappa]$, where $g_{p,q}$ is the intersection point of lines h_p and l_q. Under a general position assumption (otherwise we slightly perturb the grid position to handle degenerate cases), each unit square in any set $\mathcal{S}_i$, for $i \in [N]$ contains exactly one grid point. For each $g_{p,q}$, we store a Boolean *activity value* 1 or 0 based

on its intersection with $\mathcal{S}_i$ (for any step $i \in [N]$). If $g_{p,q}$ intersects at least one square of S_i, we say that it is *active* and set the value to 1; otherwise, we set the value to 0. Observe that for each grid point $g_{p,q}$ and each time step i at most one square of $\mathcal{S}_i$ intersecting $g_{p,q}$ can be chosen in any MAX-IS. This holds because all squares that intersect the same grid point form a clique in G_i, and at most one square from a clique can be chosen in any independent set.

We first initialize an independent set M_1 for $G_1 = (\mathcal{S}_1, E_1)$ with $|M_1| \geq |OPT_1|/4$. For each horizontal line $h_j \in H$, we compute two independent sets $M^1_{h_j}$ and $M^2_{h_j}$, where $M^1_{h_j}$ (resp. $M^2_{h_j}$) contains an arbitrary square intersecting each odd (resp. even) grid point on h_j. Since every other grid point is omitted in these sets, any two selected squares are independent. Let $M(h_j) = \arg\max\{|M^1_{h_j}|, |M^2_{h_j}|\}$ be the larger of the two independent sets. We define $p(h_j) = |M^1_{h_j}|$ and $q(h_j) = |M^2_{h_j}|$, as well as $c(h_j) = |M(h_j)| = \max\{p(h_j), q(h_j)\}$.

We construct the independent sets $M(E_H) = \bigcup_{j=1}^{\lfloor \kappa/2 \rfloor}(M(h_{2j}))$ for E_H and $M(O_H) = \bigcup_{j=1}^{\lfloor \kappa/2 \rfloor}(M(h_{2j-1})$ for O_H. We return $M_1 = \arg\max\{|M(E_H)|, |M(O_H)|\}$ as the independent set for G_1. See Figure 2 for an illustration. The initialization of all $O(\kappa^2)$ variables and the computation of the first set M_1 take $O(\kappa^2)$ time. (Alternatively, a hash table would be more space efficient, but could not provide the $O(1)$-update time guarantee.)

▶ **Lemma 5 (⋆).** *The set M_1 is an independent set of $G_1 = (\mathcal{S}_1, E_1)$ with $|M_1| \geq |OPT_1|/4$ and can be computed in $O(\kappa^2)$ time.*

In the following step, when we move from G_i to G_{i+1}, for any $i \in [N]$, a square s_x is inserted into $\mathcal{S}_i$ or deleted from $\mathcal{S}_i$. Intuitively, we check the activity value of the grid point that s_x intersects. If the update has no effect on its activity value, we keep $M_{i+1} = M_i$. Otherwise, we update the activity value, the corresponding cardinality counters, and report the solution accordingly. All of these operations can be performed in $O(1)$-time. For a more detailed description see the full version of this paper.

▶ **Lemma 6 (⋆).** *The set M_i is an independent set of $G_i = (\mathcal{S}_i, E_i)$ for each $i \in [N]$ and $|M_i| \geq |OPT_i|/4$.*

Lemmas 5 and 6 together with the $O(1)$ update time yield:

▶ **Theorem 7 (⋆).** *We can maintain a 4-approximate maximum independent set in a dynamic unit square intersection graph, deterministically, in $O(1)$ update time.*

4.2 $2(1 + \frac{1}{k})$-Approximation Algorithm with $O(k)$ Update Time

Next, we improve the approximation factor from 4 to $2(1 + \frac{1}{k})$, for any $k \geq 1$, by combining the shifting technique [29] with the insights gained from Section 4.1. This comes at the cost of an increase of the update time to $O(k^2 \log n)$, which illustrates the trade-off between solution quality and update time. We reuse the grid partition and some notations from Section 4.1. We first describe how to obtain a solution M_1 for the initial graph G_1 that is of size at least $|OPT_1|/2(1 + \frac{1}{k})$ and then discuss how to maintain this under dynamic updates.

Let $h_j \in H$ be a horizontal stabbing line and let $\mathcal{S}(h_j) \subseteq \mathcal{S}$ be the set of squares stabbed by h_j. Since they are all stabbed by h_j, the intersection graph of $\mathcal{S}(h_j)$ is equivalent to the unit interval intersection graph obtained by projecting each unit square $s_x \in \mathcal{S}(h_j)$ to a unit interval i_x on the line h_j; we denote this set of unit intervals as $I(h_j)$. First, we sort the intervals in $I(h_j)$ from left to right. Next we define $k+1$ *groups* with respect to h_j that are formed by deleting those squares and their corresponding intervals from $S(h_j)$ and $I(h_j)$, respectively, that intersect every $k+1$-th grid point on h_j, starting from some $g_{j,\alpha}$ with $\alpha \in [k+1]$. Now consider the k consecutive grid points on h_j between two deleted grid points

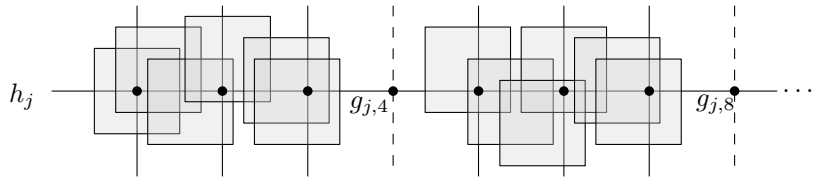


Figure 3 Illustration of a group on line h_j for $k = 3$ with the two subgroups $I_1^3(h_j)$ and $I_5^3(h_j)$.

in one such group, say, $\{g_{j,\ell}, \ldots, g_{j,\ell+k-1}\}$ for some $\ell \in [\kappa]$. Let $I_\ell^k(h_j) \subseteq I(h_j)$ be the set of unit intervals intersecting the k grid points $g_{j,\ell}$ to $g_{j,\ell+k-1}$. We refer to them as *subgroups*. See Figure 3 for an illustration. Observe that the maximum size of an independent set of each subgroup is at most k, since the width of each subgroup is strictly less than $k+1$ and each interval has unit length.

We compute M_1 for G_1 as follows. For each stabbing line $h_j \in H$, we form the $k+1$ different groups of $I(h_j)$. For each group, a MAX-IS is computed optimally and separately inside each subgroup. Since any two subgroups are horizontally separated and thus independent, we can then take the union of the independent sets of the subgroups to get an independent set for the entire group. This is done with the linear-time greedy algorithm to compute maximum independent sets for interval graphs [26]. Let $\{M_{h_j}^1, \ldots, M_{h_j}^{k+1}\}$ be $k+1$ maximum independent sets for the $k+1$ different groups and let $M(h_j) = \arg\max\{|M_{h_j}^1|, |M_{h_j}^2|, \ldots, |M_{h_j}^{k+1}|\}$ be one with maximum size. We store its cardinality as $c(h_j) = \max\{|M_{h_j}^i| \mid i \in [k+1]\}$. Next, we compute an independent set for E_H, denoted by $M(E_H)$, by composing it from the best solutions $M(h_j)$ from the even stabbing lines, i.e., $M(E_H) = \bigcup_{j=1}^{\lfloor \kappa/2 \rfloor} M(h_{2j})$ and its cardinality $|M(E_H)| = \sum_{j=1}^{\lfloor \kappa/2 \rfloor} c(h_{2j})$. Similarly, we compute an independent set for O_H as $M(O_H) = \bigcup_{j=1}^{\lfloor \kappa/2 \rfloor} M(h_{2j-1})$ and its cardinality $|M(O_H)| = \sum_{j=1}^{\lfloor \kappa/2 \rfloor} c(h_{2j-1})$. Finally, we return $M_1 = \arg\max\{|M(E_H)|, |M(O_H)|\}$ as the solution for G_1.

▶ **Lemma 8** (⋆). *The independent set M_1 of $G_1 = (\mathcal{S}_1, E_1)$ can be computed in $O(n \log n + kn)$ time and $|M_1| \geq |OPT_1|/2(1 + \frac{1}{k})$.*

Next, we describe a pre-processing step, which is required for the dynamic updates.
PRE-PROCESSING: For each horizontal line $h_j \in H$, consider a group. For each subgroup $I_\ell^k(h_j)$ (for some $\ell \in [k+1]$), we construct a balanced binary tree $T(I_\ell^k(h_j))$ storing the intervals of $I_\ell^k(h_j)$ in left-to-right order (indexed by their left endpoints) in the leaves. This process is done for each group of every horizontal line $h_j \in H$. We mark those leaves in $T(I_\ell^k(h_j))$ as *selected* that correspond to an independent interval in the solution and maintain a list of pointers to those independent intervals. This tree also lets us quickly identify the location of an interval that is inserted or deleted. In fact, while we run the greedy algorithm on $I_\ell^k(h_j)$, we can already mark precisely the selected intervals for the independent set.

When we perform the update step from $G_i = (\mathcal{S}_i, E_i)$ to $G_{i+1} = (\mathcal{S}_{i+1}, E_{i+1})$, either a square is inserted into $\mathcal{S}_i$ or deleted from $\mathcal{S}_i$. Let s_x and i_x be this square and its corresponding interval. Let $g_{u,v}$ (for some $u, v \in [\kappa]$) be the grid point that intersects s_x. We describe here the INSERTION and refer to the full version of this paper for the DELETION.
INSERTION: The insertion of i_x affects all but one of the groups on line h_u. We describe the procedure for one such group on h_u; it is then repeated for the other groups. In each group, i_x appears in exactly one subgroup and the other subgroups remain unaffected. This subgroup, say $I_\ell^k(h_u)$, is determined by the index v of the grid point $g_{u,v}$ intersecting i_x. First, we locate i_x in the sorted list of intervals of $I_\ell^k(h_u)$, which can be done in $O(\log n)$ time by searching in the associated tree $T(I_\ell^k(h_u))$. If i_x is immediately left of a selected interval

i_y, but does not intersect the previous selected interval, then i_x becomes a new selected interval that replaces i_y and triggers a sequence of updates of the later selected intervals. Let us first consider the case that i_x is not selected as a new independent interval. Then we simply insert i_x into $T(I_\ell^k(h_u))$ in $O(\log n)$ time. Otherwise, we mark i_x as selected, remove the selection mark from its successor interval i_y, and replace i_y by i_x in the maintained subsolution. Since the right endpoint of i_x is left of the right endpoint of i_y, this change possibly triggers a sequence of updates to the subsequent selected intervals.

We thus identify in $T(I_\ell^k(h_u))$ the leftmost interval i_z that starts to the right of the right endpoint of i_x. This takes $O(\log n)$ time. If this interval i_z is not yet marked as selected, we replace the previous successor of i_x in the current list of selected intervals by i_z and repeat the update process for i_z. Otherwise, if i_z is already selected, we can stop the update of the subsolution as there would be no further changes.

Since a maximum independent set in each subgroup contains at most k intervals, the update time is $O(k \log n)$ per group and $O(k^2 \log n)$ for all k affected groups.

While doing the updates, we collect the new selected intervals as the MAX-IS for the subgroup $I_\ell^k(h_u)$. For all groups affected by the insertion of i_x we update the corresponding independent sets $M_{h_u}^p$ for $p \in [k+1]$, whenever some updates of selected intervals were necessary. Then we select the largest independent set of all $k+1$ groups as $M(h_j)$ and update its new cardinality in $c(h_j)$. Finally, we update the independent sets $M(E_H)$ and $M(O_H)$ and their cardinalities and return $M_{i+1} = \arg\max\{|M(E_H)|, |M(O_H)|\}$ as the solution for G_{i+1}.

▶ **Lemma 9** (⋆). *The set M_i is an independent set of $G_i = (\mathcal{S}_i, E_i)$ for each $i \in [N]$ and $|M_i| \geq |OPT_i|/2(1 + \frac{1}{k})$.*

With Lemma 9 and the update time discussion in the full version of the paper we obtain:

▶ **Theorem 10** (⋆). *We can maintain a $2(1 + \frac{1}{k})$-approximate maximum independent set in a dynamic unit square intersection graph, deterministically, in $O(k^2 \log n)$ update time.*

4.3 2-Approximation Algorithm with $O(\omega \log n)$ Update Time

We finally design a 2-approximation algorithm for the MAX-IS problem on dynamic axis-aligned unit height, but arbitrary width rectangles. Let $\mathcal{B}$ be the bounding box of the dynamic set of rectangles $\widetilde{\mathcal{R}} = \bigcup_{i \in [N]} \mathcal{R}_i$. We begin by dividing $\mathcal{B}$ into horizontal strips of height 1 defined by the set $H = \{h_1, \ldots, h_\kappa\}$ of $\kappa = O(n)$ horizontal lines. We assume, w.l.o.g., that every rectangle in $\widetilde{\mathcal{R}}$ is stabbed by exactly one line in H. For a set of rectangles $\mathcal{R}$, we denote the subset stabbed by a line h_j as $\mathcal{R}(h_j) \subseteq \mathcal{R}$.

We first describe how to obtain an independent set M_1 for the initial graph $G_1 = (\mathcal{R}_1, E_1)$ such that $|M_1| \geq |OPT_1|/2$ by using the following algorithm of Agarwal et al. [3]. For each horizontal line $h_j \in H$, we compute a maximum independent set for $\mathcal{R}_1(h_j)$. The set $\mathcal{R}_i(h_j)$ (for any $i \in [N]$ and $j \in [\kappa]$) can again be seen as an interval graph. For a set of n intervals, a MAX-IS can be computed by a left-to-right greedy algorithm visiting the intervals in the order of their right endpoints in $O(n \log n)$ time. So for each horizontal line $h_j \in H$, let $M(h_j)$ be a MAX-IS of $\mathcal{R}_1(h_j)$, and let $c(h_j) = |M(h_j)|$. Then we construct the independent set $M(E_H) = \bigcup_{j=1}^{\lfloor \kappa/2 \rfloor}(M(h_{2j}))$ for E_H. Similarly, we construct the independent set $M(O_H) = \bigcup_{j=1}^{\lfloor \kappa/2 \rfloor}(M(h_{2j-1})$ for O_H. We return $M_1 = \arg\max\{|M(E_H)|, |M(O_H)|\}$ as the independent set for $G_1 = (\mathcal{R}_1, E_1)$. See Figure 4 for an illustration.

▶ **Lemma 11** (Theorem 2, [3]). *The set M_1 is an independent set of $G_1 = (\mathcal{R}_1, E_1)$ with $|M_1| \geq |OPT_1|/2$ and can be computed in $O(n \log n)$ time.*

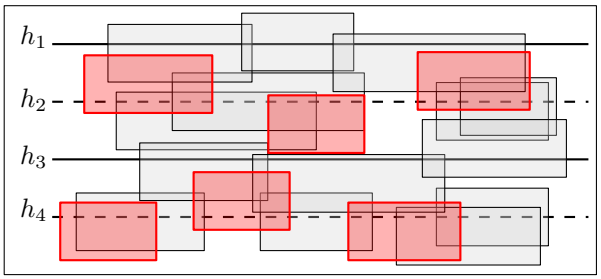


Figure 4 Example instance with four horizontal lines. Red rectangles represent the computed 2-approximate solution, which here is $M(E(H))$.

We describe the following pre-processing step to initialize in $O(n \log n)$ time the data structures that are required for the subsequent dynamic updates.
Pre-Processing: Consider a stabbing line h_j and the stabbed set of rectangles $\mathcal{R}_i(h_j)$ for some $i \in [N]$. We denote the corresponding set of intervals as $I(h_j)$. We build a balanced binary search tree $T_l(I(h_j))$, storing the intervals in $I(h_j)$ in left-to-right order based on their left endpoints. This is called the *left tree* of $I(h_j)$. We augment the left tree such that each tree node additionally stores a pointer to the interval with leftmost right endpoint in its subtree. This pointer structure can easily be computed by a bottom-up pass through $T_l(I(h_j))$. Note that a leaf update in $T_l(I(h_j))$ takes $O(\log n)$ time as for standard binary search trees, but we can in the same $O(\log n)$ time propagate the change that potentially affects the leftmost right endpoints of the tree nodes along the path to the root. Additionally, we store the set of selected independent intervals for the Max-IS of $I(h_j)$ in left-to-right order in another balanced binary search tree $T_s(I(h_j))$ (the so-called *solution tree*). Let ω_j be the cardinality of the maximum independent of $I(h_j)$ for $j \in [\kappa]$, and let $\omega = \max_j \omega_j$ be the maximum of these cardinalities over all stabbing lines h_j.

When we move from G_i to G_{i+1} (for some $1 \leq i < N$), either we insert a new rectangle into $\mathcal{R}_i$ or delete one rectangle from $\mathcal{R}_i$. Let r_x be the rectangle that is inserted or deleted, let i_x be its corresponding interval, and let h_j (for some $j \in [\kappa]$) be the horizontal line that intersects r_x. In what follows, we describe how to maintain a 2-approximate Max-IS with $O(\omega_j \log n) = O(\omega \log n)$ update time. We distinguish Insertion and Deletion.
Insertion: We first determine whether i_x should be a new selected interval or not. Because the greedy algorithm for constructing the Max-IS visits the intervals in left-to-right order based on the right endpoints, we need to reconstruct the state of the algorithm when it would visit i_x. We query the solution tree $T_s(I(h_j))$ with both endpoints of i_x in $O(\log \omega_j)$ time. If and only if both search paths end up between the same two leaves belonging to two consecutive selected intervals i_y and i_z (considering their right endpoints), then i_x would have been chosen as the next selected interval after i_y and before i_z in the greedy algorithm. This implies that i_y and i_x are independent, but i_x and i_z may or may not intersect.

If the two search paths in the solution tree are different, then i_x does not become a new selected interval, and we simply insert it into $T_l(I(h_j))$ in $O(\log n)$ time. Else we also insert i_x into $T_l(I(h_j))$, but we also have to perform a sequence of selection update operations, which are more involved for intervals of arbitrary length compared to the updates in Section 4.2. Figure 5 shows an example. First we mark i_x as selected and insert it into $T_s(I(h_j))$. Now we need to identify the next selected interval right of i_x that would have been found by the greedy algorithm. We use the left tree $T_l(I(h_j))$ to search in $O(\log n)$ time for the interval i'_z with leftmost right endpoint, whose left endpoint is right of the right endpoint p of i_x.

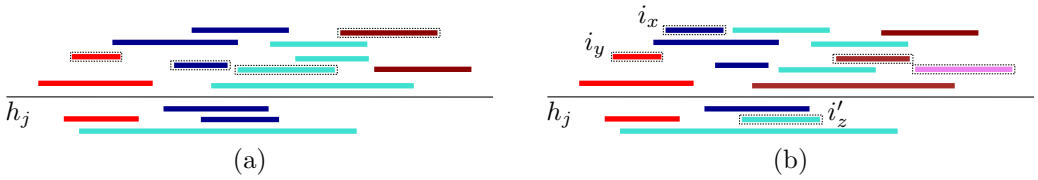


Figure 5 Illustration of the updates triggered by the insertion of an interval; selected independent intervals are marked by a dotted bounding box and intervals intersected by a selected interval have the same color as the rightmost such interval. (a) Before insertion of i_x, (b) after insertion of i_x.

More precisely, we search for p in $T_l(I(h_j))$ and whenever the search path branches into the left subtree, we compare whether the leftmost right endpoint stored in the root of the right subtree is left of the right endpoint of the current candidate interval. Once a leaf is reached, the leftmost found candidate interval is the desired interval i'_z. This interval i'_z is precisely the first interval after i_x in the order considered by the greedy algorithm that is independent of i_x and thus must be the next selected interval. If $i_z \neq i'_z$, we repeat the update process for i'_z as if it would have been the newly inserted interval until we reach the end of $I(h_j)$; otherwise we keep i_z as the successor of i_x and stop the update process.

For each update of a selected interval, we perform one search in $T_l(I(h_j))$ in $O(\log n)$ time. There are at most ω_j updates, so the update time is $O(\omega_j \log n)$. Finally, we need to delete $O(\omega_j)$ old selected intervals from and insert $O(\omega_j)$ new selected intervals into the solution tree $T_s(I(h_j))$, which takes $O(\log \omega_j)$ time for each insertion and deletion. We now re-evaluate the new Max-IS $M(h_j)$ and its cardinality $c(h_j)$, which possibly affects $M(E_H)$ or $M(O_H)$. We obtain the new independent set $M_{i+1} = \arg\max\{|M(E_H)|, |M(O_H)|\}$ for $G_{i+1} = (\mathcal{R}_{i+1}, E_{i+1})$.

Deletion: If the interval i_x to be deleted is not a selected interval, it is sufficient to delete it from the left tree $T_l(I(h_j))$ in $O(\log n)$ time. Otherwise, if i_x is a selected interval, let i_y be the selected interval preceding i_x in the solution tree $T_s(I(h_j))$. We first delete i_x from $T_l(I(h_j))$ and $T_s(I(h_j))$. Then we need to select a new interval to replace i_x according to the greedy Max-IS algorithm, which is the interval i_z whose right endpoint is leftmost among all intervals that are completely to the right of i_y. We find this interval i_z again by a search in the left tree $T_l(I(h_j))$ with the right endpoint of i_y as the query point. We make i_z a new selected interval and use the right endpoint of i_z as the query point for finding the next selected interval in $T_l(I(h_j))$. We repeat this process until we have reached the last interval of $I(h_j)$.

As for the Insertion step, each update of a selected interval requires $O(\log n)$ time due to the query for the next selected interval in $T_l(I(h_j))$. There are $O(\omega_j)$ such updates. Further, we need to update the solution tree $T_s(I(h_j))$ by performing $O(\omega_j)$ insertions and deletions of seeds, each in $O(\log \omega_j)$ time. Once all updates to the selected intervals and the data structures for $I(h_j)$ are done, we re-evaluate the new Max-IS $M(h_j)$ and its cardinality $c(h_j)$, which possibly affects $M(E_H)$ or $M(O_H)$. This yields the new independent set $M_{i+1} = \arg\max\{|M(E_H)|, |M(O_H)|\}$ for $G_{i+1} = (\mathcal{R}_{i+1}, E_{i+1})$.

▶ **Lemma 12.** *The set M_i is an independent set of $G_i = (\mathcal{R}_i, E_i)$ for each $i \in [N]$ and $|M_i| \geq |OPT_i|/2$.*

Proof. We prove the lemma by induction. From Lemma 11 we know that M_1 satisfies the claim, and in particular each set $M(h)$ for $h \in H$ is a Max-IS of the interval set $I(h)$. So let us consider the set M_i for $i \geq 2$ and assume that M_{i-1} satisfies the claim by the induction hypothesis. Let r_x and i_x be the updated rectangle and its interval, and assume that it

belongs to the stabbing line h_j. Then we know that for each $h_k \in H$ with $k \neq j$ the set $M(h_k)$ is not affected by the update to r_x and thus is a MAX-IS by the induction hypothesis. It remains to show that the update operations described above restore a MAX-IS $M(h_j)$ for the set $I(h_j)$. But in fact the updates are designed in such a way that the resulting set of selected intervals is identical to the set of intervals that would be found by the greedy MAX-IS algorithm for $I(h_j)$. Therefore $M(h_j)$ is a MAX-IS for $I(h_j)$ and by the pigeonhole principle $|M_i| \geq |OPT_i|/2$. ◀

Each update of a rectangle r_x (and its interval i_x) triggers either an INSERTION or a DELETION operation on the unique stabbing line of r_x. As we have argued in the description of these two update operations, the insertion or deletion of i_x requires one $O(\log n)$-time update in the left tree data structure. If i_x is a selected independent interval, the update further triggers a sequence of at most ω_j selection updates, each of which requires $O(\log n)$ time. Hence the update time is bounded by $O(\omega_j \log n) = O(\omega \log n)$. Recall that ω_j and ω are output-sensitive parameters describing the maximum size of an independent set of $I(h)$ for a specific stabbing line $h = h_j$ or any stabbing line h.

▶ **Theorem 13.** *We can maintain a 2-approximate maximum independent set in a dynamic unit-height arbitrary-width rectangle intersection graph, deterministically, in $O(\omega \log n)$ time, where ω is the cardinality of a maximum independent set of the rectangles stabbed by the horizontal stabbing line affected by the dynamic update.*

▶ Remark 14. We note that Gavruskin et al. [24] gave a dynamic algorithm for maintaining a MAX-IS on *proper* interval graphs. Their algorithm runs in amortized time $O(\log^2 n)$ for insertion and deletion, and $O(\log n)$ for element-wise decision queries. The complexity to report a MAX-IS J is $\Theta(|J|)$. Whether the same result holds for general interval graphs was posed as an open problem [24]. Our algorithm in fact solves the MAX-IS problem on arbitrary dynamic interval graphs, which is of independent interest. Moreover, it explicitly maintains a MAX-IS at every step.

5 Experiments

We implemented all our MAX-IS approximation algorithms presented in Sections 3 and 4 in order to empirically evaluate their trade-offs in terms of *solution quality*, i.e., the cardinality of the computed independent sets, and *update time* measured on a set of suitable synthetic and real-world map-labeling benchmark instances with unit squares. The goal is to identify those algorithms that best balance the two performance criteria. Moreover, for smaller benchmark instances with up to 2 000 squares, we compute exact MAX-IS solutions using a MaxSAT model by Klute et al. [32] that we solve with MaxHS 3.0 (see `www.maxhs.org`). These exact solutions allow us to evaluate the optimality gaps of the different algorithms in light of their worst-case approximation guarantees. Finally, we investigate the speed-ups gained by using our dynamic update algorithms compared to the baseline of recomputing new solutions from scratch with their respective static algorithm after each update.

5.1 Experimental Setup

We have implemented the five algorithms (and their greedy augmentation variants) listed below in C++. The experiments were run on a server equipped with two Intel Xeon E5-2640 v4 processors (2.4 GHz 10-core) and 160GB RAM. The machine ran the 64-bit version of Ubuntu Bionic (18.04.2 LTS). The code was compiled with g++ 7.4.0.

MIS-graph A naive graph-based dynamic MIS algorithm, explicitly maintaining the square intersection graph and a MIS [4, Sec. 3]. In order to evaluate and compare the performance of our algorithm *MIS-ORS* (Section 3) for the MIS problem, we have implemented this alternative dynamic algorithm. This algorithm, instead of maintaining the current instance in a dynamic geometric data structure, maintains the rectangle intersection graph explicitly as a baseline approach. We use standard adjacency lists to represent the intersection graph, implemented as unordered sets in C++. Now, to obtain a MIS at the first step, we add an arbitrary (unmarked) vertex v to the solution and mark $N(v)$ in the corresponding intersection graph. This process is repeated iteratively until there is no unmarked vertex left in the intersection graph. Clearly, by following this greedy method, we obtain a MIS. Moreover, for each vertex v, we maintain an augmenting *counter* that stores the number of vertices from its neighborhood $N(v)$ that are contained in the current MIS.

This approach handles the updates in a straightforward manner. When a new vertex is inserted, its corresponding rectangle introduces new intersections in the current intersection graph. Therefore, when adding this vertex, we also determine the edges that are required to be added to the intersection graph. Notice that, unlike the canonical vertex update operation defined in the literature, where the adjacencies of the new vertex are part of the dynamic update, here, we actually need to figure out the neighborhood of a vertex. This is done by iterating over each vertex and checking whether its corresponding rectangle is overlapping with the newly inserted rectangle. Thus, it takes $O(n)$ time to obtain the neighborhood of this vertex. If the newly inserted rectangle has no intersection with any rectangle from the current solution, then we simply add its vertex to the solution; otherwise, we ignore it. Finally, we update the counters. If a vertex is deleted, we update the intersection graph by deleting its corresponding rectangle. If the deleted vertex was in the solution, then we decrease the counters of its neighbors by 1. Once the counter of a vertex is updated to 0, we add this vertex into the solution. Both the insertion (after computing $N(v)$) and deletion operation for a vertex v take $O(\deg(v))$ time each to update the intersection graph and the MIS solution.

MIS-ORS The dynamic MIS algorithm based on orthogonal range searching (Section 3); this algorithm provides a 4-approximation. In the implementation we used the dynamic orthogonal range searching data structure implemented in CGAL (version 4.11.2), which is based on a dynamic Delaunay triangulation [35, Chapter 10.6]. Hence, this implementation does not provide the sub-logarithmic worst-case update time of Theorem 3.

grid The grid-based 4-approximation algorithm (Section 4.1).

grid-k The shifting-based $2(1+\frac{1}{k})$-approximation algorithm (Section 4.2). In the experiments we use $k = 2$ (i.e., a 3-approximation) and $k = 4$ (i.e., a 2.5-approximation).

line The stabbing-line based 2-approximation algorithm (Section 4.3).

Since the algorithms *grid*, *grid-k*, and *line* are based on partitioning the set of squares and considering only sufficiently segregated subsets, they produce a lot of white space in practice. For instance, they ignore the squares stabbed by either all the even or all the odd stabbing lines completely in order to create isolated subinstances. In practice, it is therefore interesting to augment the computed approximate MAX-IS by greedily adding independent, but initially discarded squares. We have also implemented the greedy variants of these algorithms, which are denoted as *g-grid*, *g-grid-k*, and *g-line*.

We created three types of benchmark instances. The synthetic data sets consist of n 30×30-pixel squares placed inside a bounding rectangle $\mathcal{B}$ of size $1\,080 \times 720$ pixels, which also creates different densities. The real-world instances use the same square size, but geographic feature distributions. For the updates we consider three models: *insertion-only*, *deletion-only*, and *mixed*, where the latter selects insertion or deletion uniformly at random.

Table 1 Specification of the six OSM instances.

	post-CH	peaks-AT	hotels-CH	hotels-AT	peaks-CH	hamlets-CH
features (n)	646	652	1 788	2 209	4 320	4 326
overlaps (m)	5 376	5 418	28 124	68 985	107 372	159 270
density (m/n)	8.32	8.31	15.73	31.23	24.85	36.92

Gaussian In the Gaussian model, we generate n squares randomly in $\mathcal{B}$ according to an overlay of three Gaussian distributions, where 70% of the squares are from the first distribution, 20% from the second one, and 10% from the third one. The means are sampled uniformly at random in $\mathcal{B}$ and the standard deviation is 100 in both dimensions.

Uniform In the uniform model, we generate n squares in $\mathcal{B}$ uniformly at random.

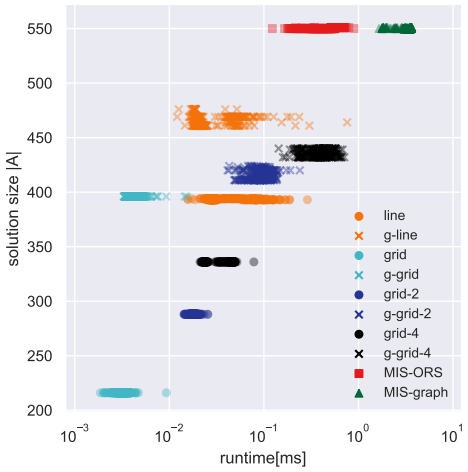
Real-world We created six real-world data sets by extracting point features from OpenStreetMap (OSM), see Table 1 for their detailed properties.

5.2 Experimental Results

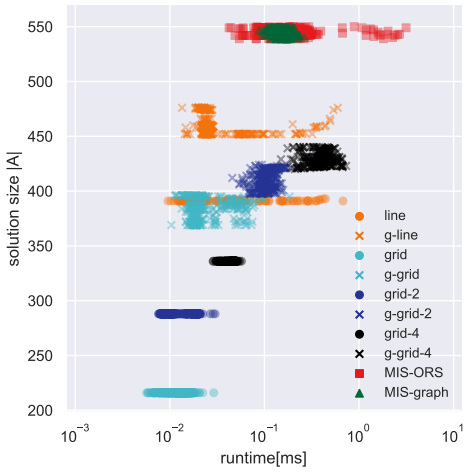
Time-quality trade-offs. For our first set of experiments we compare the five implemented algorithms, including their greedy variants, in terms of update time and size of the computed independent sets. Figure 6 shows scatter plots of runtime vs. solution size on uniform and Gaussian benchmarks, where algorithms with dots in the top-left corner perform well in both measures.

We first consider the results for the uniform instances with $n = 10\,000$ squares in the top row of Figure 6. Each algorithm performed $N = 400$ updates, either insertions (Figure 6a) or deletions (Figure 6b) and each update is shown as one point in the respective color. Both plots show that the two MIS algorithms compute the best solutions with almost the same size and well ahead of the rest. While *MIS-ORS* is clearly faster than *MIS-graph* on insertions, they are comparably fast for deletions, with some slower outliers of *MIS-ORS*. The approximation algorithms *grid*, *grid*-2, *grid*-4, and *line* (without the greedy optimizations) show their predicted relative behavior: The better the solution quality, the worse the update times. Algorithms *line* and *g-line* show a wide range of update times, spanning almost two orders of magnitude. Adding the greedy optimization drastically improves the solution quality in all cases, but typically at the cost of higher runtimes. For *g-grid-k* the algorithms get slower by an order of magnitude and increase the solution size by 30–50%. For *g-grid*, the additional runtime is not as significant (but deletions are slower than insertions), and the solution size almost doubles. Finally, *g-line* is nearly as fast as *line*, and reaches the best quality among the approximation algorithms with about 80% of the MIS solutions, but faster by one or two orders of magnitude.

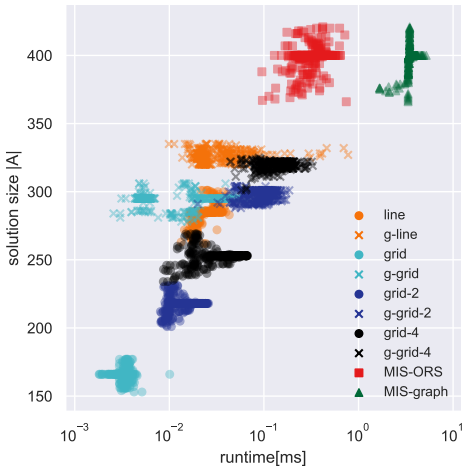
For the results of the Gaussian instances with $n = 10\,000$ squares and $N = 400$ updates plotted in Figures 6c (insertions) and 6d (deletions) we observe the same ranking between the different algorithms. However, due to the non-uniform distribution of squares, the solution sizes are more varying, especially for the insertions. For the deletions it is interesting to see that *grid* and *MIS-graph* have more strongly varying runtimes, which is in contrast to the deletions in the uniform instance, possibly due to the dependence on the vertex degree. The best solutions are computed by *MIS-ORS* and *MIS-graph*, which show similar deletion times, but the insertion times of *MIS-ORS* are one order of magnitude faster than *MIS-graph*. Algorithm *g-line* again reaches more than 80% of the quality of the MIS algorithms, with a speed-up between one and two orders of magnitude.



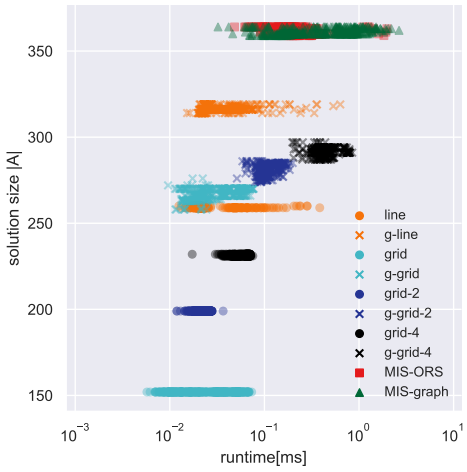
(a) Uniform, $n = 10\,000$, 400 insertions.



(b) Uniform, $n = 10\,000$, 400 deletions.



(c) Gaussian, $n = 10\,000$, 400 insertions.



(d) Gaussian, $n = 10\,000$, 400 deletions.

Figure 6 Time-quality scatter plots for synthetic benchmark instances. The x-axis (log-scale) shows runtime, the y-axis shows the solution size.

Optimality gaps. Next, let us look at the results of the real-world instances in Figure 7. The first four instances in Figure 7a–d, were small enough so that we could compute each Max-IS exactly with MaxHS and compare the solutions of the approximation algorithms with the optimum on the y-axis. The largest two instances in Figure 7e and 7f plot the solution size on the y-axis. First, let us consider Figure 7c as a representative, which is based on a data set of 1 788 hotels and hostels in Switzerland with mixed updates of 10% of the squares ($N = 179$). Generally speaking, the results of the different algorithms are much more overlapping in terms of quality than for the synthetic instances. The plot shows that the MIS algorithms reach consistently between 80% and 85% of the optimum, but are sometimes outperformed by *g-grid*-4 and *g-line*. Regarding the runtime, *MIS-ORS* has more homogeneous update times ranging between the extrema of *MIS-graph*, which suffers from the rather slow insertions. The original approximations are well above their respective worst-case ratios, but stay between 45% and 65% of the optimum. The greedy extensions

push this towards larger solutions, at the cost of higher runtimes. However, *g-line* seems to provide a very good balance between quality and speed. We point out that because the updates comprise insertions and deletions, the marks for algorithms that are sensitive to the update type, such as *g-grid* and *MIS-graph* form two separate runtime clusters. The same relative observations of the algorithms' performance can be made in Figures 7a–d, yet they show different absolute quality offsets and variance.

Let us next consider the largest OSM instance in Figure 7f. It again reflects the same findings as obtained from the smaller instances. The instance consists of $n = 4\,326$ hamlets in Switzerland with 10% mixed updates ($N = 433$) and is denser by a factor of about 2.3 than hotels-CH (see Table 1). There is quite some overlap of the different algorithms in terms of the solution size, yet the algorithms form the same general ranking pattern as observed before. Interestingly, while the MIS algorithms contribute some of the best solutions, they also show a variance of ± 50 squares. In contrast, *g-line*, the best of the approximation algorithms, is competing well and is more stable in terms of solution size and again about an order of magnitude faster than the MIS algorithms. The update-type dependent behavior of *MIS-graph* with its significantly slower insertions is observed once more, making *MIS-ORS* the better choice for a mixed update model.

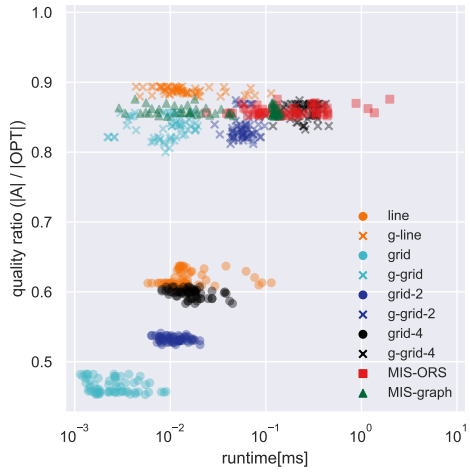
Finally, Figure 8 shows the optimality ratios of the algorithms for small uniform and Gaussian instances with $n = 1\,000$ squares. They confirm our earlier observations, but also show that for these small instances, *MIS-graph* is about as fast as *MIS-ORS* for insertions and faster than *MIS-ORS* for deletions. This is because the graph size and vertex degrees do not yet influence the running time of *MIS-graph* strongly. Yet, as the next experiment shows, this changes drastically, as the instance size grows.

Runtimes. In our last experiment, we explore in more detail the scalability of the algorithms for larger instances, both relative to each other and in comparison to the re-computation times of their corresponding static algorithms. We generated one random instance with $n = 1\,000k$ squares for each $k \in \{1, 2, 4, 8, 16, 32\}$ and measured the average update times over $n/10$ insertions or deletions. The results for the Gaussian and uniform model are plotted in Figure 9. Considering the update times for insertions, we confirm the observations from the scatter plots in terms of the performance ranking. Most algorithms grow only very slowly in terms of their running time, with the notable exception of *MIS-graph*, but that was to be expected. For deletions, *MIS-graph* is initially faster than *MIS-ORS*, but again shows the steepest increase in runtime. Deletions in the Gaussian model also affect the runtime of *grid* and *g-grid* quite noticeably, yet one order of magnitude below *MIS-graph*.

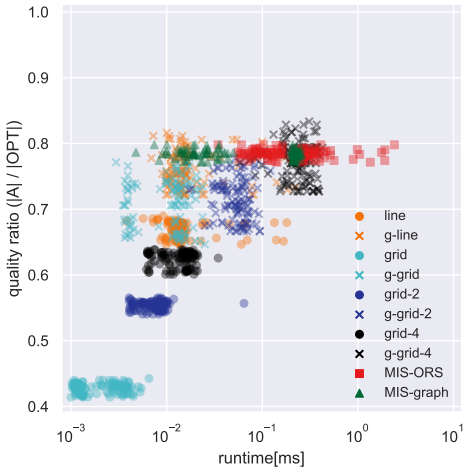
In the comparison with their non-dynamic versions, i.e., re-computing solutions after each update, the dynamic algorithms indeed show a significant speed-up in practice, already for small instance sizes of $n = 1\,000$, and even more so as n grows (notice the different y-offsets). For some algorithms, including *MIS-ORS* and *g-line*, this can be as high as 3–4 orders of magnitude for $n = 32\,000$. It clearly confirms that the investigation of algorithms for dynamic MIS and Max-IS problems for rectangles is well justified also from a practical point of view.

5.3 Discussion

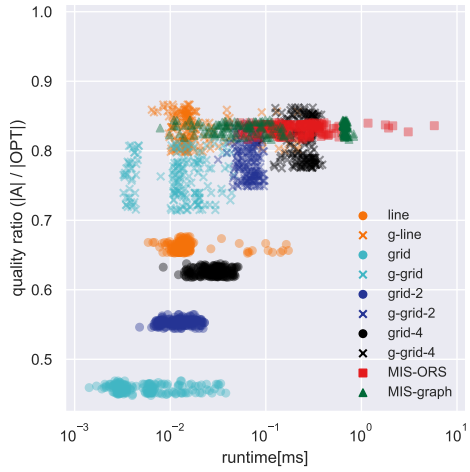
Our experimental evaluation provides several interesting insights into the practical performance of the different algorithms. First of all, both MIS-based algorithms generally showed the best solution quality in the field, reaching 85% of the exact Max-IS size, where we could compare against optimal solutions. This is in strong contrast to their factor-4 worst-case approximation guarantee of only 25%.



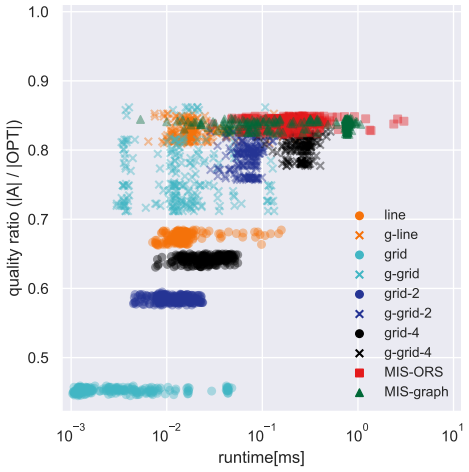
(a) post-CH, 10% mixed updates.



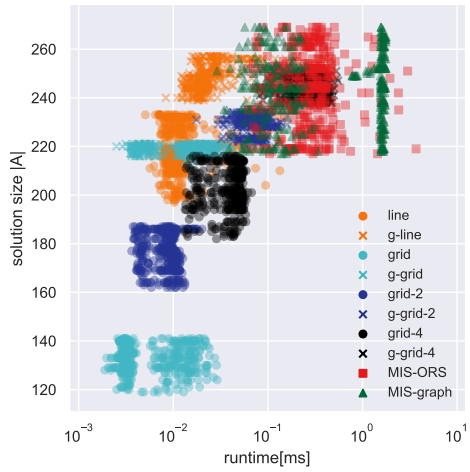
(b) peaks-AT, 10% mixed updates.



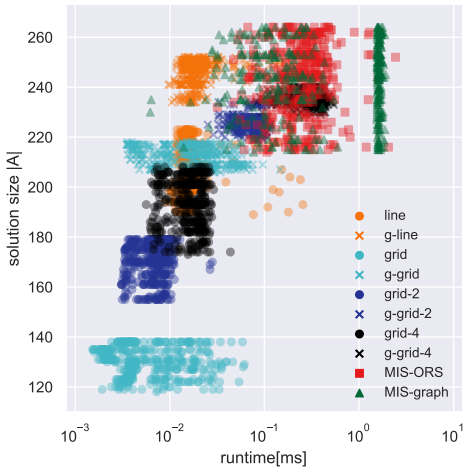
(c) hotels-CH, 10% mixed updates.



(d) hotels-AT, 10% mixed updates.

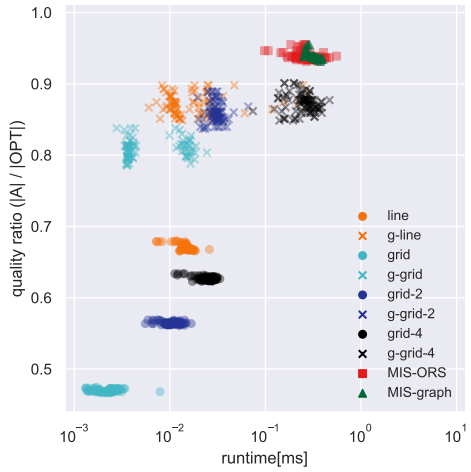


(e) peaks-CH, 10% mixed updates.

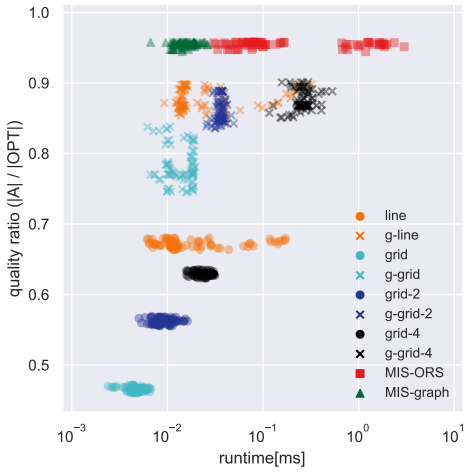


(f) hamlets-CH, 10% mixed updates.

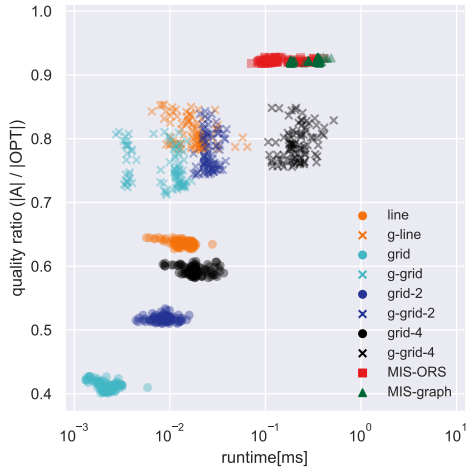
Figure 7 Time-quality scatter plots for the OSM instances. The x-axis (log-scale) shows runtime. The y-axis shows the quality ratio compared to an optimal MAX-IS solution for the smaller instances (a)–(d), and the solution size for the larger instances (e)–(f).



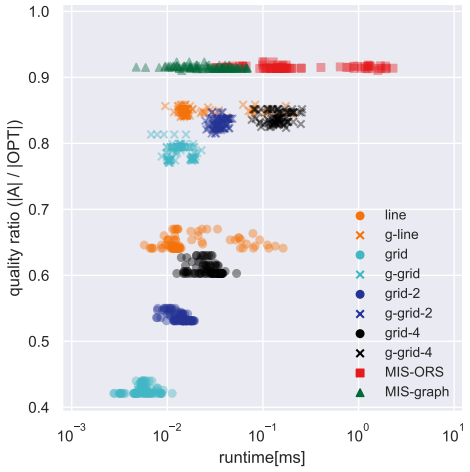
(a) Uniform, 100 insertions.



(b) Uniform, 100 deletions.



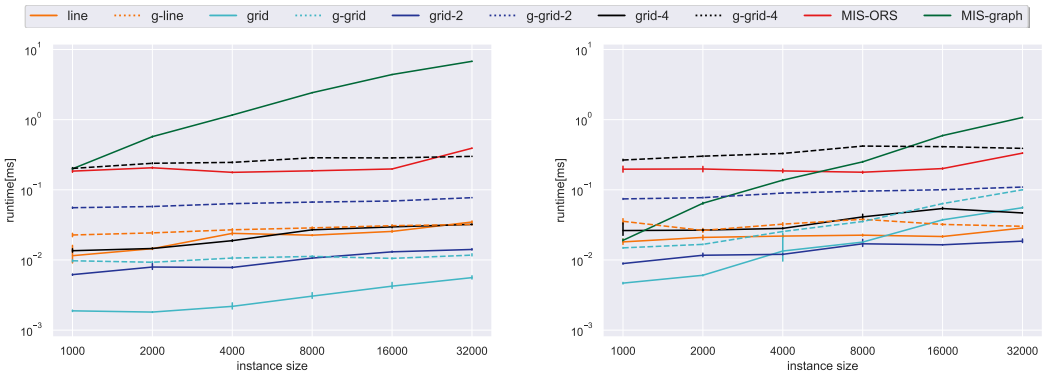
(c) Gaussian, 100 insertions.



(d) Gaussian, 100 deletions.

Figure 8 Time-quality scatter plots for uniform and Gaussian instances with $n = 1\,000$ squares. The x-axis (log-scale) shows runtime. The y-axis shows the quality ratio compared to an optimal Max-IS solution.

Our algorithm *MIS-ORS* avoids storing the intersection graph explicitly. Instead, we only store the relevant geometric information in a dynamic data structure and derive edges on demand. Therefore it breaks the natural barrier of $\Omega(\Delta)$ (amortized) vertex update in a dynamic graph, where Δ is the maximum degree in the graph. However, it has to find the intersections using the complex range query, which takes $O(\log n)$ time. We did not involve any geometric data structure in the baseline MIS approach *MIS-graph*. Recall that, the update of the intersection graph when adding a new rectangle includes the time to figure out the neighborhood of the newly added vertex. Therefore, the graph-based algorithm showed a slow insertion update and was quite sensitive to the size of instances in insertion updates. However, the deletion update only depends on the degree of the involved vertex, not the size of instances directly. And as expected, the graph-based algorithm was indeed much faster for small instances, but *MIS-ORS* was more scalable in our experiment. However, the intersection graph update for *MIS-graph* can be improved by using additionally a geometric



(a) Update times for insertions (Gaussian).

(b) Update times for deletions (Gaussian).

(c) Re-computation times for insertions (Gaussian).

(d) Re-computation times for deletions (Gaussian).

(e) Update times for insertions (uniform).

(f) Update times for deletions (uniform).

(g) Re-computation times for insertions (uniform).

(h) Re-computation times for deletions (uniform).

Figure 9 Log-log runtime plots (notice the different y-offsets) for dynamic updates and re-computation on Gaussian instances (a)–(d) and uniform instances (e)–(h) of size $n = 1\,000$ to $32\,000$, averaged over $n/10$ updates. Error bars indicate the standard deviation.

data structure to store the rectangle set and detect intersections. We expect that it would show improvements for insertion updates, but may slow down deletions, since the state-of-the art data structure provides only an amortized update time guarantee. Therefore, it is an open question whether the performance of *MIS-graph* can indeed be improved by using a suitable dynamic geometric data structure. Note, *MIS-ORS* too, can sometimes show slower deletions, due to the necessary complex orthogonal range search in some cases. Recall that, in our implementation, we used a dynamic range searching data structure from CGAL, which does not provide the theoretical sub-logarithmic worst-case update time of Chan et al. [14] used in Theorem 3. Exploring how *MIS-ORS* can benefit from such a state-of-the-art dynamic data structure in practice remains to be investigated in future work. Notwithstanding, it remains to state that even with the suboptimal data structure, *MIS-ORS* was able to compute its solutions for up to 32 000 squares in less than 1ms. So if solution quality is the priority, then the *MIS-ORS* algorithm is the method of choice. It provides the best solutions (together with *MIS-graph*), but is significantly more scalable.

An expected observation is that while consistently exceeding their theoretical guarantees, the approximation algorithms do not perform too well in practice due to their pigeonhole choice of too strictly separated subinstances. However, a simple greedy augmentation of the approximate solutions can boost the solution size significantly, and for some algorithms even to almost that of the MIS algorithms. Of course, at the same time this increases the runtime of the algorithms. We want to point out *g-line*, the greedy-augmented version of the 2-approximation algorithm *line*, as it computes very good solutions, even comparable or better than *MIS-ORS* and *MIS-graph* for the real-world instances, and at 80% of the MIS solutions for the synthetic instances. At the same time, *g-line* is still significantly faster than *MIS-ORS* and *MIS-graph* and thus turns out to be a well-balanced compromise between time and quality. It would be our recommended method if *MIS-ORS* or *MIS-graph* are too slow for an application.

6 Conclusions

We investigated the MIS and Max-IS problems on dynamic sets of uniform rectangles and uniform-height rectangles from an algorithm engineering perspective, providing both theoretical results for maintaining a MIS or an approximate Max-IS and reporting insights from an experimental study. Open problems for future work include (i) finding Max-IS sublinear-update-time approximation algorithms for dynamic unit squares with approximation ratio better than 2, (ii) studying similar questions for dynamic disk graphs, and (iii) implementing improvements such as a sub-logarithmic dynamic range searching data structure to speed-up our algorithm *MIS-ORS*. Moreover, it would be interesting to design dynamic approximation schemes for Max-IS that maintain stability in a solution.

References

1 Amir Abboud, Raghavendra Addanki, Fabrizio Grandoni, Debmalya Panigrahi, and Barna Saha. Dynamic set cover: improved algorithms and lower bounds. In *Symposium on Theory of Computing (STOC'19)*, pages 114–125. ACM, 2019. doi:10.1145/3313276.3316376.

2 Anna Adamaszek and Andreas Wiese. Approximation schemes for maximum weight independent set of rectangles. In *Foundations of Computer Science (FOCS'13)*, pages 400–409. IEEE, 2013. doi:10.1109/FOCS.2013.50.

3 Pankaj K Agarwal, Marc Van Kreveld, and Subhash Suri. Label placement by maximum independent set in rectangles. *Computational Geometry*, 11(3-4):209–218, 1998. doi:10.1016/S0925-7721(98)00028-5.

4 Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear update time. In *Symposium on Theory of Computing (STOC'18)*, pages 815–826, 2018. doi:10.1145/3188745.3188922.

5 Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear in n update time. In *Symposium on Discrete Algorithms (SODA'19)*, pages 1919–1936. SIAM, 2019. doi:10.1137/1.9781611975482.116.

6 Egon Balas and Chang Sung Yu. Finding a maximum clique in an arbitrary graph. *SIAM Journal on Computing*, 15(4):1054–1068, 1986. doi:10.1137/0215075.

7 Ken Been, Martin Nöllenburg, Sheung-Hung Poon, and Alexander Wolff. Optimizing active ranges for consistent dynamic map labeling. *Comput. Geom. Theory Appl.*, 43(3):312–328, 2010. doi:10.1016/j.comgeo.2009.03.006.

8 Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Cliff Stein, and Madhu Sudan. Fully dynamic maximal independent set with polylogarithmic update time. In *Foundations of Computer Science (FOCS'19)*, pages 382–405. IEEE, 2019. doi:10.1109/FOCS.2019.00032.

9 Aaron Bernstein, Sebastian Forster, and Monika Henzinger. A deamortization approach for dynamic spanner and dynamic maximal matching. In *Symposium on Discrete Algorithms (SODA'19)*, pages 1899–1918. SIAM, 2019. doi:10.1137/1.9781611975482.115.

10 Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika Henzinger. Deterministic fully dynamic approximate vertex cover and fractional matching in $o(1)$ amortized update time. In *Integer Programming and Combinatorial Optimization (IPCO'17)*, volume 10328 of *LNCS*, pages 86–98. Springer, 2017. doi:10.1007/978-3-319-59250-3_8.

11 Sayan Bhattacharya, Deeparnab Chakrabarty, Monika Henzinger, and Danupon Nanongkai. Dynamic algorithms for graph coloring. In *Symposium on Discrete Algorithms (SODA'18)*, pages 1–20. SIAM, 2018. doi:10.1137/1.9781611975031.1.

12 Parinya Chalermsook and Julia Chuzhoy. Maximum independent set of rectangles. In *Symposium on Discrete Algorithms (SODA'09)*, pages 892–901. SIAM, 2009. doi:10.1137/1.9781611973068.97.

13 Timothy M Chan and Sariel Har-Peled. Approximation algorithms for maximum independent set of pseudo-disks. *Discrete & Computational Geometry*, 48(2):373–392, 2012. doi:10.1007/s00454-012-9417-5.

14 Timothy M. Chan and Konstantinos Tsakalidis. Dynamic orthogonal range searching on the RAM, revisited. In Boris Aronov and Matthew J. Katz, editors, *Computational Geometry (SoCG'17)*, volume 77 of *LIPIcs*, pages 28:1–28:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.SoCG.2017.28.

15 Shiri Chechik and Tianyi Zhang. Fully dynamic maximal independent set in expected poly-log update time. In *Foundations of Computer Science (FOCS'19)*, pages 370–381. IEEE, 2019. doi:10.1109/FOCS.2019.00031.

16 Julia Chuzhoy and Alina Ene. On approximating maximum independent set of rectangles. In *Foundations of Computer Science (FOCS'16)*, pages 820–829. IEEE, 2016. doi:10.1109/FOCS.2016.92.

17 Graham Cormode, Jacques Dark, and Christian Konrad. Independent sets in vertex-arrival streams. In *International Colloquium on Automata, Languages, and Programming (ICALP'19)*, volume 132 of *LIPIcs*, pages 45:1–45:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.ICALP.2019.45.

18 David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Dynamic graph algorithms. In Mikhail J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 8. CRC Press, 1999. doi:10.1.1.43.8372.

19 Thomas Erlebach, Klaus Jansen, and Eike Seidel. Polynomial-time approximation schemes for geometric intersection graphs. *SIAM Journal on Computing*, 34(6):1302–1323, 2005. doi:10.1137/s0097539702402676.

20 Michael Formann and Frank Wagner. A packing problem with applications to lettering of maps. In *Symposium on Computational Geometry (SoCG'91)*, pages 281–288. ACM, 1991. doi:10.1145/109648.109680.

21 Robert J. Fowler, Mike Paterson, and Steven L. Tanimoto. Optimal packing and covering in the plane are NP-complete. *Inf. Process. Lett.*, 12(3):133–137, 1981. doi:10.1016/0020-0190(81)90111-3.

22 Edith Gabriel. Spatio-temporal point pattern analysis and modeling. In Shashi Shekhar, Hui Xiong, and Xun Zhou, editors, *Encyclopedia of GIS*, pages 1–8. Springer, 2015. doi:10.1007/978-3-319-23519-6_1646-1.

23 Buddhima Gamlath, Michael Kapralov, Andreas Maggiori, Ola Svensson, and David Wajc. Online matching with general arrivals. In *Foundations of Computer Science (FOCS'19)*, pages 26–37, 2019. doi:10.1109/FOCS.2019.00011.

24 Alexander Gavruskin, Bakhadyr Khoussainov, Mikhail Kokho, and Jiamou Liu. Dynamic algorithms for monotonic interval scheduling problem. *Theoretical Computer Science*, 562:227–242, 2015. doi:10.1016/j.tcs.2014.09.046.

25 Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter. Consistent labeling of rotating maps. *J. Computational Geometry*, 7(1):308–331, 2016. doi:10.20382/jocg.v7i1a15.

26 U. I. Gupta, D. T. Lee, and Joseph Y.-T. Leung. Efficient algorithms for interval graphs and circular-arc graphs. *Networks*, 12(4):459–467, 1982. doi:10.1002/net.3230120410.

27 Johan Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182(1):105–142, 1999. doi:10.1007/BF02392825.

28 Monika Henzinger, Stefan Neumann, and Andreas Wiese. Dynamic approximate maximum independent set of intervals, hypercubes and hyperrectangles. In Sergio Cabello and Danny Z. Chen, editors, *Symposium on Computational Geometry (SoCG 2020)*, volume 164 of *LIPIcs*, pages 51:1–51:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.SoCG.2020.51.

29 Dorit S. Hochbaum and Wolfgang Maass. Approximation schemes for covering and packing problems in image processing and vlsi. *J. ACM*, 32(1):130–136, 1985. doi:10.1145/2455.214106.

30 John E Hopcroft and Richard M Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973. doi:10.1137/0202019.

31 Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller, J. W. Thatcher, and J. D. Bohlinger, editors, *Complexity of Computer Computations*, pages 85–103, 1972. doi:10.1007/978-1-4684-2001-2_9.

32 Fabian Klute, Guangping Li, Raphael Löffler, Martin Nöllenburg, and Manuela Schmidt. Exploring semi-automatic map labeling. In *Advances in Geographic Information Systems (SIGSPATIAL'19)*, pages 13–22. ACM, 2019. doi:10.1145/3347146.3359359.

33 Nathan Linial. Distributive graph algorithms global solutions from local data. In *Foundations of Computer Science (SFCS'87)*, pages 331–335. IEEE, 1987. doi:10.1109/SFCS.1987.20.

34 Kurt Mehlhorn and Stefan Näher. Dynamic fractional cascading. *Algorithmica*, 5(1–4):215–241, 1990. doi:10.1007/BF01840386.

35 Kurt Mehlhorn and Stefan Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999. doi:10.1145/204865.204889.

36 Huy N Nguyen and Krzysztof Onak. Constant-time approximation algorithms via local improvements. In *Foundations of Computer Science (FOCS'08)*, pages 327–336. IEEE, 2008. doi:10.1109/FOCS.2008.81.

37 Panos M Pardalos and Jue Xue. The maximum clique problem. *Journal of Global Optimization*, 4(3):301–328, 1994. doi:10.1007/BF01098364.

38 René van Bevern, Matthias Mnich, Rolf Niedermeier, and Mathias Weller. Interval scheduling and colorful independent sets. *Journal of Scheduling*, 18(5):449–469, 2015. doi:10.1007/s10951-014-0398-5.

39 Marc J. van Kreveld, Tycho Strijk, and Alexander Wolff. Point set labeling with sliding labels. In Ravi Janardan, editor, *Proceedings of the Fourteenth Annual Symposium on Computational Geometry, Minneapolis, Minnesota, USA, June 7-10, 1998*, pages 337–346. ACM, 1998. doi:10.1145/276884.276922.

40 Frank Wagner and Alexander Wolff. A practical map labeling algorithm. *Comput. Geom. Theory Appl.*, 7:387–404, 1997. doi:10.1016/S0925-7721(96)00007-7.