Dynamic Algorithms in Unit Disk Graphs

DALGO Lab

Pohang University of Science and Technology

November 16, 2023

Abstract

This research focuses on developing dynamic algorithms in unit disk graph for efficiently solving graph-based problems. First, we will present an efficient algorithm for the vertex cover problem. In our algorithm, updates require O(1) time, and we guarantee $2^{O(\sqrt{k})}$ time for queries.

1 Introduction

The time complexity of the vertex cover problem for planar graphs is known as $2^{O(\sqrt{k})}n^{O(1)}$ [1]. In our approach, we obtain the time complexity for vertex cover query in unit disk graph as $2^{O(\sqrt{k})}$.

We employ a grid wherein each cell's diagonal is of unit length, thereby ensuring that all vertices in a single cell constitute a clique. When analyzing a given cell, 5×5 grid centered on that cell will be used. The term "cell around" and "neighbors" refer to the adjacent 5×5 grid of cells surrounding a given cell. Cells are named with distinct names based on V_C, which is number of vertices in the cell C.

1. if V_C is greater than 2, $friendly\,cell$
2. if $V_C = 1$, $lonely\,cell$
3. if $V_C = 0$, $empty\,cell$

2 Algorithm

In this section, we will explain our data structure and the algorithms for updates and queries.

2.1 Data structure

In order to dynamically update the graph, we will preserve this data.

1. Kernel for the original graph
2. for each cell C, all the vertices in this cell and V_C
3. the size of the kernel, that is the number of vertices in the kernel

In the kernel, we have $friendly\,cells$, $lonely\,cells$ where at least one of neighbors is $friendly\,cell$, and not isolated $lonely\,cells$.

2.2 Insertion

In this section, we will show how to maintain the kernel during the vertex insertion process.

First, update the vertices in the cell which contains the new vertex V_{new} and increase V_C of this cell by 1. Check the neighbors that have become *lonely cells* from *isolated lonely cells* due to the insertion and if so, include them in the kernel. Also, increase the size of the kernel by the number of cells included. If V_C is greater than 3, increase the kernel size by 1. This cell is already included in the kernel. If V_C is equal to 2, check the isolation of this cell before insertion. If this cell was isolated, include this cell in the kernel and increase the kernel size by 2. If this cell was not isolated, this cell is already in the kernel and increase the kernel size by 1. Next, now this cell is no longer a lonely cell, include all the neighbors in the kernel and increase the kernel size. If V_C is equal to 1, check the neighbors and if at least one cell is *friendly cell*, include this cell in the kernel and increase the kernel size by 1. If all the surrounding cells are *lonely cells* and *empty cells*, determine whether this cell is isolated. If the vertex is isolated, do nothing. Else if the vertex is not isolated, include the cell in the kernel and increase the kernel size by 1.

2.3 Deletion

In this section, we will show how to maintain the kernel during the vertex deletion process.

First, update the vertices in the cell which contains the deleted vertex V_{del} and decrease V_C by 1. If V_C is greater than 1, decrease the kernel size by 1. This cell is still *friendly cell*, so neighbors should be included in the kernel as before. Otherwise, determine the cells that has become *isolated lonely cells* by the deletion from *lonely cells* and remove the cells from the kernel and decrease the kernel size. If V_C is equal to 1, check the neighbors and if at least one cell is *friendly cell*, this cell is still kept in the kernel as before. So, decrease the kernel size by 1. If all the neighbors are *lonely cells* and *empty cells*, determine whether this cell is isolated. If the cell is isolated, remove this cell from the kernel. So, decrease the kernel size by 2. Else if the vertex is not isolated, this cell is still kept in the kernel as before. So, decrease the kernel size by 1. If V_C is equal to 0, determine whether this cell was in the kernel before deletion. If it was in the kernel, remove this cell from the kernel reduce the kernel size by 1 and. Otherwise, do nothing.

2.4 Query

For the vertex cover query, algorithm operates as follows.

If the kernel size is greater than $O(k)$, the k-vertex cover is impossible. Else, vertex cover algorithm is performed on the kernel that we maintain.

3 Correctness

We need to make sure that the vertex cover of the kernel is same as the vertex cover of the original graph. We also prove the bounds on the number of vertices in the kernel and analyze its time complexity.

3.1 Correctness of the kernel

We have four kinds of edges: edges within the same cell, between *friendly cells*, between *friendly cells* and *lonely cells*, between *lonely cells*. Because all the vertices in *friendly cells* are included in the kernel, all edges between vertices in the same cell is covered. Similary,

edges between *friendly cells* are all included in the kernel. Edges between *friendly cells* and *lonely cells* are covered by including lonely cells which have friendly cells as neighbors in the kernel. Edges between lonely cells are covered because isolated vertex in *lonely cells* were removed from the kernel and the rest were included in the kernel.

3.2 Kernel size restriction

First, we will show that if the vertex cover is possible, the size of kerenel is $O(k)$.

By summing up the number of the vertices in *friendly cells* and *lonely cells* in the kernel, we can get total number of vertices in the kernel. For each *friendly cell* C, $V_C - 1$ vertices must be included in the vertex cover. Let the set of *friendly cells* be $\mathcal{F}$. For k-vertex cover, the size of vertex cover is less than or equal to k. So for all *friendly cells*, $\sum_{\mathcal{F}} V_C - |\mathcal{F}| \leq k$. Using the fact that $\mathcal{F}$ is less than $k+1$, the number of vertices in all *friendly cells* is less than $2k+1$.

Lonely cells can be separated into two types depending on the presence of *friendly cells* around. The number of *friendly cells* is less than or equal to k, and each *friendly cell* can only cover constant number of *lonely cells*, so the number of *lonely cells* which have at leat one *friendly cell* as neighbors is $O(k)$. *Lonely cell* that are surrounded by only *lonely cells* or *empty cells* has constant number of edge. In this case, there can be up to $O(k)$ edges between lonely cell if k-vertex is possible, so the number of vertices in *lonely cell* is $O(k)$.

Through the above two processes, both of *friendly cells* and *lonely cells* in the kernel contain $O(k)$ vertices if k-vertex cover is possible.

3.3 Time complexity

In the update process, dealing with V_C and kernel size takes $O(1)$ time. Determining cell isolation takes $O(1)$ time because we only need to consider constant number of lonely cells around it. Similary, finding the *friendly cells* and *lonely cells* around the cell also takes constant time. So each update takes $O(1)$ time.

In 3.2 we show that the total number of vertices in the kernel is $O(k)$ if k-vertex cover is possible. So $2^{O(\sqrt{k})}$ is guaranteed for each query.

References

[1] Erik D Demaine, Fedor V Fomin, Mohammadtaghi Hajiaghayi, and Dimitrios M Thilikos. Subexponential parameterized algorithms on bounded-genus graphs and h-minor-free graphs. *Journal of the ACM (JACM)*, 52(6):866–893, 2005.