5.1.1 Finding the min-degree vertex

For some earlier algorithms, we needed to find a vertex of minimum degree while the graph is changing. For example, consider the greedy-algorithm to approximate VertexCover (and more generally for SetCover). Here we always want to find a vertex of minimum degree, but then the graph changes because we delete this vertex and its incident edges. In its most general form, we want to support the query operation

- *findMinDegree*: returns a vertex that has minimum degree in the current graph (ties are broken arbitrarily)

during the following update operations:

- *addIsolatedVertex(v)*: adds to G a new vertex v that has degree 0.
- *addEdge(v, w)*: adds to G a new edge (v, w) (it is assumed that v, w already exist).
- *deleteEdge(v, w)*: removes edge (v, w) (it is assumed that (v, w) already exists).
- *deleteIsolatedVertex(v)*: removes vertex v (it is assumed that v already exists and has degree 0).

(To avoid trivialities, let us say that the empty graph has minimum and maximum degree 0.) We assume that the graph is stored in such a way that the four update-operations take $O(1)$ time; this is possible using suitably cross-linked vertex-lists and edge-lists and incidence-lists (details are left as an exercise).

The brute-force approach for *findMinDegree* would be to compute the vertex of minimum degree from scratch, i.e., to go through all vertices, look up their degrees, and retain the one where the degree is smallest. This would take $\Theta(n)$ time every time, making the algorithm slow. (For example, the greedy-algorithm for VertexCover would take $\Theta(nm)$ time with this approach.) Can we be faster?

Using a priority queue

A first idea is to use a priority queue, with the degree of each vertex as a key. Recall that a priority queue is a data structure that stores items with keys (typically assumed to be numbers, but they can be anything that can be compared and they do not have to be distinct). It supports the operations *insert*, *findMin*, *deleteMin*, and some realizations also support *increaseKey* and *decreaseKey*.

We can use a priority queue Q for finding the vertex of minimum degree as follows, by storing every vertex v in Q with key $\deg(v)$. The operations can be implemented as follows:

- *findMinDegree*: execute Q.*findMin*.
- *addIsolatedVertex(v)*: execute Q.*insert* with v and key 0.

- *addEdge*(v, w): execute Q.*increaseKey* twice to increase the degree of v and w by one each.

- *deleteEdge*(v, w): execute Q.*decreaseKey* twice to decrease the degree of v and w by one each.

- *deleteIsolatedVertex*(v): execute Q.*decreaseKey* to decrease the key of v to -1 and then do Q.*deleteMin*

If we realize the priority queue with a binary heap, and let each vertex store a reference to where it is in Q, then all priority-queue operations take $O(\log n)$ time. (Here n is the number of vertices in the graph, which is the same as the number of items stored in the priority queue.) Each graph-update or graph-query operation uses a constant number of priority-queue operations, and so also takes $O(\log n)$ time.

One could observe that many operations could be made faster. For example, *addEdge* actually only takes $O(1)$ time, because it increases the keys by only one, and (as one can argue) this takes $O(1)$ time in binary heaps where keys are integers. There are also other priority queue implementations (e.g. Fibonacci-heaps) that implement *decreaseKey* in $O(1)$ (amortized) time for any key. However, no comparison-based implementation can do both *insert* and *deleteMin* in $o(\log n)$ time (else we could sort faster).

One could also observe that all our keys are integers in the range $[0, n]$, and there are priority-queue implementations (such as van Emde Boas trees) that take advantage of this. Using them, one could achieve a run-time of $O(\log \log n)$ for the above operations. We will not pursue this further because in our special situation (where all key-changes are ± 1) we can create a simpler data structure that achieves an even better run-time.

Using buckets

The data structure to use here is hashing with chaining, using $h(k) = k$ (i.e., direct addressing) as the hash-function. This sometimes is also called a *bucket-structure* (the familiar bucket-sorting algorithm is based on the same idea). We will review what it means here without using hashing-terminology.

The main data structure is a dynamic array B that stores lists of vertices. Each vertex v is stored in the list $B[\deg(v)]$. We also keep a reference δ to the current minimum degree in the graph. See Figure 5.1. Initially (when the graph is empty) we have $\delta = 0$.

We are assuming that each v stores (and hence can look up in constant time) its degree $\deg(v)$ as well as where it is in $B[\deg(v)]$, i.e. it has a reference to the appropriate list-item. Initially, when the graph With this, we can now implement four of the required operations easily as follows:

- *GetMinDegVertex()*: Simply return the first vertex in $B[\delta]$.

- *AddIsolatedVertex*(v): Simply insert v in $B[0]$. Also, the minimum degree now is 0, so update $\delta \leftarrow 0$.

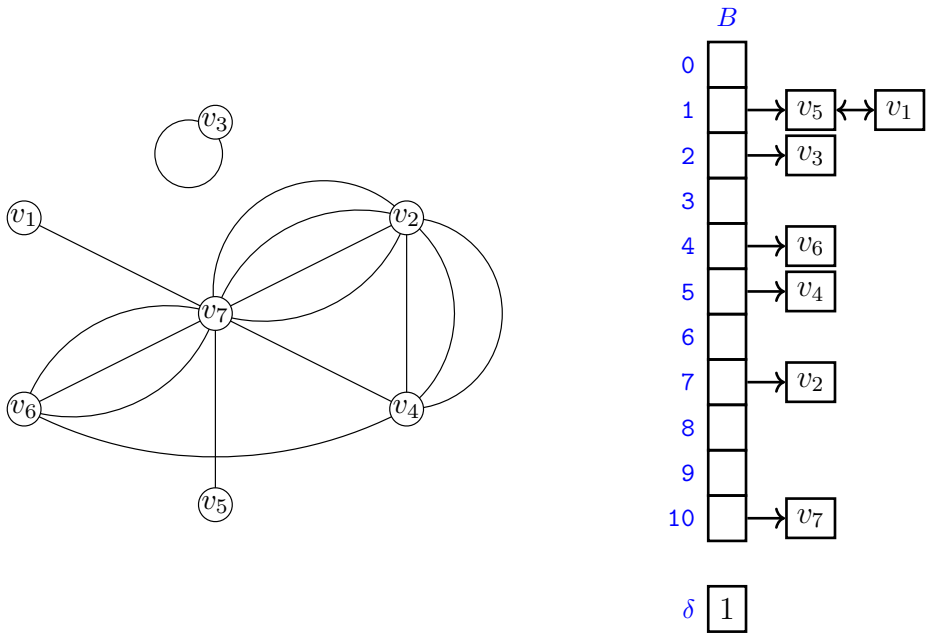

Figure 5.1: A graph and where its vertices are in the bucket structure.

For this and other graph-update operations, we list here only the changes that need to be done to the bucket-structure; obviously we also need to do some changes to the data structure that stores the graph.

Also, during this and other graph-update operations, the degree of v and where it is in the bucket-structure changes. Since we store this information with v, we must update it as it changes. However, these updates should be straightforward, and so we will not clutter our pseudocodes with them.

- *AddEdge*(v, w): Call *IncreaseDegree* for each of v and w. Here, *IncreaseDegree*(v) receives as input a vertex that is already in the bucket-structure, and it increases its degree by one. To do so, look up $d \leftarrow \deg(v)$, and look up the list item ℓ that stores v in bucket $B[d]$. Remove ℓ from $B[d]$ and insert it into $B[d{+}1]$. Finally we must check whether this has affected the minimum degree, which can happen only if $\delta{=}\, d$. If so, then check whether $B[d]$ is now empty; if it is then update $\delta \leftarrow d{+}1$.

- *DeleteEdge*(v, w): Call *DecreaseDegree* for each of v and w. Here, *DecreaseDegree*(v) is symmetric to *IncreaseDegree*: look up d and ℓ and move ℓ into $B[d{-}1]$. Also update $\delta \leftarrow \min\{\delta, d{-}1\}$.

Directly from the description it should be clear that we spend $O(1)$ time for each of the above 'easy' operations. This leaves only one operation: *DeleteIsolatedVertex*(v). This is much harder, because we know that prior to the deletion the minimum degree was 0, but unless there is another isolated vertex, the minimum degree now increases and we have

no idea what it is going to be. So finding it in $O(1)$ worst-case time seems hard, and the following algorithm uses more time in the worst-case.

Algorithm 6: *DeleteIsolatedVertex(v)*

```
1 ℓ ← item where v is stored in B[0]
2 delete ℓ from B[0]
3 if there are vertices left then
4     while (B[δ] is empty) do δ++                    // min degree increases
```

The run-time of this algorithm is $O(1 + \delta_{\text{after}})$. In the worst case, this is very bad. However, δ_{after} can be big only if a lot of edges are left, which means that we did a lot of (cheap) edge-insertions earlier. So we make edge-insertions 'pay for' the expense that now happens when we delete an isolated vertices. Put different: we should use amortized analysis.

We will use the potential function method. Assume that time units are chosen such that all easy operations take at most one time unit, and *DeleteIsolatedVertex* takes at most $1 + \delta_{\text{after}}$ time units. Define the potential function to be

$$\Phi(i) = 2m - \delta,$$

which is non-negative since $\delta \leq \sum_{v \in V} \deg(v) = 2m$. Assuming that we start with an initially empty graph, we have $\Phi(0) = 0 + 0 = 0$, so this is a valid potential function.

If $\mathcal{O}$ is one of the 'easy' operations, then $m_{\text{after}} \leq m_{\text{before}} + 1$ and $\delta_{\text{after}} \geq \delta_{\text{before}} - 2$, so $\Phi_{\text{after}} \leq \Phi_{\text{before}} + 4$ (this is actually an over-estimate, but good enough). So the amortized run-time is

$$T^{actual}(\mathcal{O}) + \Phi_{\text{after}} - \Phi_{\text{before}} \leq 1 + 4 \in O(1).$$

Now consider operation *DeleteIsolatedVertex*. Its actual run-time is at most $1 + \delta_{\text{after}}$ time units. We also know that $m_{\text{after}} = m_{\text{before}}$ and $\delta_{\text{before}} = 0$. Therefore

$$\begin{aligned} T^{actual}(\mathcal{O}) + \Phi_{\text{after}} - \Phi_{\text{before}} &\leq (1 + \delta_{\text{after}}) + (2m_{\text{after}} - \delta_{\text{after}}) - (2m_{\text{before}} - \delta_{\text{before}}) \\ &= 1 + (\underbrace{2m_{\text{after}} - 2m_{\text{before}}}_{0}) + \underbrace{\delta_{\text{before}}}_{0} = 1 \end{aligned}$$

So the amortized time for all operations is in $O(1)$.