Algorithm 6.6 The inner loop subroutine `build_grad`($\mathsf{V}, \mathcal{G}, \mathcal{G}'$, `grad_table`) of the back-propagation algorithm, called by the back-propagation algorithm defined in algorithm 6.5.

Require: V, the variable whose gradient should be added to $\mathcal{G}$ and `grad_table`.
Require: $\mathcal{G}$, the graph to modify.
Require: $\mathcal{G}'$, the restriction of $\mathcal{G}$ to nodes that participate in the gradient.
Require: `grad_table`, a data structure mapping nodes to their gradients
  if V is in `grad_table` **then**
    Return `grad_table`[V]
  end if
  $i \leftarrow 1$
  for C in `get_consumers`($\mathsf{V}, \mathcal{G}'$) **do**
    `op` $\leftarrow$ `get_operation`(C)
    $\mathsf{D} \leftarrow$ `build_grad`($\mathsf{C}, \mathcal{G}, \mathcal{G}'$, `grad_table`)
    $\mathsf{G}^{(i)} \leftarrow$ `op.bprop`(`get_inputs`($\mathsf{C}, \mathcal{G}'$), V, D)
    $i \leftarrow i + 1$
  end for
  $\mathsf{G} \leftarrow \sum_i \mathsf{G}^{(i)}$
  `grad_table`[V] $= \mathsf{G}$
  Insert G and the operations creating it into $\mathcal{G}$
  Return G

roughly chain-structured, causing back-propagation to have $O(n)$ cost. This is far better than the naive approach, which might need to execute exponentially many nodes. This potentially exponential cost can be seen by expanding and rewriting the recursive chain rule (equation 6.49) non-recursively:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{\substack{\text{path}\,(u^{(\pi_1)}, u^{(\pi_2)}, \ldots, u^{(\pi_t)}),\\ \text{from}\,\pi_1 = j\,\text{to}\,\pi_t = n}} \prod_{k=2}^{t} \frac{\partial u^{(\pi_k)}}{\partial u^{(\pi_{k-1})}}. \tag{6.55}$$

Since the number of paths from node j to node n can grow exponentially in the length of these paths, the number of terms in the above sum, which is the number of such paths, can grow exponentially with the depth of the forward propagation graph. This large cost would be incurred because the same computation for $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ would be redone many times. To avoid such recomputation, we can think of back-propagation as a table-filling algorithm that takes advantage of storing intermediate results $\frac{\partial u^{(n)}}{\partial u^{(i)}}$. Each node in the graph has a corresponding slot in a table to store the gradient for that node. By filling in these table entries in order,