The Table Merging Problem

Ethan Hardy

1 The problem

To start, let's define the problem we're trying to solve: we have some array of tables T, where each table $t \in T$ stores (and can only store) strings of length $t.\ell$ and contains $t.\texttt{count}$ strings. Each table uses K bytes of overhead, plus the space of the actual strings, such that the space used by a table t is equal to $K + t.\ell \cdot t.\texttt{count}$.

We want to minimize the total amount of space needed to store all of our tables, and the one operation we can perform to potentially reduce this is to take all the strings in one table t and store them in a table t' where $t.\ell < t'.\ell$; we'll just have to pad the strings that were in t with $t'.\ell - t.\ell$ dummy characters. So, the problem we're trying to solve here is: what's the optimal way to merge (or not merge) tables to maximize space saved?

2 Getting to the algorithm

To get to the core of the problem here, let's think about incentives. One fairly obvious observation is that merging t into t' saves us $K - t.\texttt{count} \cdot (t'.\ell - t.\ell)$ space; clearly if this value is positive we'd like to make this merge. So what reason would we have for not doing it?

Another observation we make is that merging t into t' doesn't affect any other merges into t'; $t'.\ell$ hasn't changed, and $t'.\texttt{size}$ doesn't affect the savings of a merge into t'. However, this merge does affect merges of ℓ-smaller tables into t. Specifically, any tables that we merged into t will be essentially merged into t', costing us more than they would have if we hadn't merged t into t' (since $t'.\ell > t.\ell$). A simpler way we can think about this is that if we merge t into t', t becomes unavailable for merging; that is, tables can't merge into t (they can still merge into t' or any other tables still available for merging, though).

To come up with a solution to a hard problem like this one, the first thing we can try and do is break it up into solvable subproblems. The most common axis along which to break down a problem is set of inputs: what if we can solve the problem for only one table, and then use that solution to solve the problem for that table plus one more table, and so on until we have the solution for all the tables? Unfortunately, this isn't so easy here: doing this would require that the solution for the first k tables doesn't depend on any tables that we might add in future iterations, but dependencies here are two-way. A table depends on larger-ℓ tables to know what merging options it has and depends on smaller-ℓ tables to know if it should be kept available for merging so that they could merge into it. So we can't easily section off some portion of the input tables that don't depend on the rest of the input tables.

To make this work, we need to remove one direction of dependency, which we can do by borrowing a trick from some other algorithms that utilize this type of input partitioning. What if we can solve the problem for some set of input *and* an additional restriction? This would mean using a 2-D array to store the results of our subproblems, where the second dimension specifies this other restriction, and this restriction constrain one side of the dependency, solving our issue. This works only if the union of all restriction values is equivalent to the universe of possible solutions (ie, for every possible solution to the original problem, one of the restrictions holds); by solving each input-partitioned subproblem for each possible value of restriction, we can be sure that whatever the true (unrestricted) solution is, it's described by one of our subproblem solutions. For example, the Bellman-Ford shortest path algorithm does this by putting a restriction on the number of edges you're allowed to travel.

The dependency we're going to try and constrain is the dependency of larger-ℓ tables on smaller-ℓ tables; that is, the one which motivates us to keep a larger-ℓ table unmerged to reap the benefits of merging smaller-ℓ tables into it. We'll do this by restricting which tables can be merged and which must be left unmerged; this way we don't have to depend on smaller-ℓ tables. This is because they would only influence our decision as to whether or not to leave a given table available for merging; a decision which our restriction will have already made for us.

One small issue: ideally, we'd like to have our set of possible restrictions be linear in the input since we'll have to solve a subproblem for each possible restriction, but for n tables we have 2^n possible configurations of tables to keep available. However, we can eliminate a large number of these with a simple observation: smaller-ℓ tables don't care about the full mapping of larger-ℓ table to whether that table is available for merging, they only care what the smallest ℓ value is out of all tables available for merging.

For example, suppose we have three tables $t_{\ell=4}, t_{\ell=6}, t_{\ell=8}$, define A as the set of tables which are available to be merged into, and are deciding what to do with a lower-ℓ table $t_{\ell=2}$. The scenario where $A = \{t_{\ell=4}, t_{\ell=6}, t_{\ell=8}\}$ is equivalent to any of $A = \{t_{\ell=4}, t_{\ell=6}\}, A = \{t_{\ell=4}, t_{\ell=8}\}, A = \{t_{\ell=4}\}$ for the purposes of our decision, since in any of these scenarios we just want to merge $t_{\ell=2}$ into $t_{\ell=4}$.

What this means is that we can define our restriction as "lowest-ℓ table available to be merged into", causing the number of possible restrictions to be linear in the input size (equal to, for that matter). We can then have the columns of our DP array represent the specific restriction we're enforcing, and have the rows of our DP array specify the set of input tables we have available. Now, let's move on to the actual algorithm.

3 The algorithm

To recap, we have some n-length array of tables T, where each table $t \in T$ stores (and can only store) strings of length $t.\ell$ and contains $t.\texttt{count}$ strings. Each table uses K bytes of overhead, plus the space of the actual strings, such that the space used by a table t is equal to $K + t.\ell \cdot t.\texttt{count}$. We want to minimize total space used.

First, we sort T by ℓ descending. Then we define a $n \times n$ array M; we'll compute the values of M such that $M[i][j]$ is the solution to the table merging subproblem where we only have the first $i+1$ elements of T and with the additional restriction that out of all the tables our solution leaves available for merging (ie, the tables that we don't merge into other tables), the one with the smallest ℓ is $T[j]$. For example, $M[2][1]$ is the solution to the table merging problem using only $T[0], T[1], T[2]$ and with the restriction that the smallest-ℓ table available for merging into is $T[1]$. Specifically, M holds the amount of space we gain as well as a list of merges describing the solution. Note that because any solution to the problem has exactly one lowest-ℓ unmerged table, by computing the solution for each possible ℓ in our input array, we cover every possible

unrestricted case.

For our base case, set $M[0][0] = \{\texttt{gains} : 0, \texttt{merges} : []\}$; if we only have one table, we gain 0 space and do no merges. Note that for any row i we only compute the values for columns $[0, 1, \cdots, i]$, since we can't have the lowest-ℓ unmerged table be a table we don't have as part of the subproblem. So, our first row is complete.

Then, for each row $0 < i < n$, we compute values as a function of the previous row. For each column $0 \leq j < i$, we set

$$M[i][j].\texttt{gains} = M[i-1][j].\texttt{gains} + K - (T[j].\ell - T[i].\ell) \cdot T[i].\texttt{count}$$

$$M[i][j].\texttt{merges} = M[i-1][j].\texttt{merges} + [(i, j)]$$

Let's break these down: for column j, $T[j]$ must be the unmerged table with the smallest ℓ value, and since $j < i$, $T[i].\ell < T[j].\ell$ (since T is sorted by ℓ descending) so $T[i]$ must be merged into something. Since we know that $T[j]$ is the lowest-ℓ table available for merging into, the optimal solution given our restriction is to merge $T[i]$ into $T[j]$. As for the rest of the tables ($T[0]$ to $T[i-1]$) in our subproblem, we already know that the optimal configuration for them under the same restriction is described by $M[i-1][j]$. The only change we're making is adding a table with smaller ℓ, which doesn't affect the merges we've already done with the other tables (since they can't merge into the lower-ℓ table anyways).

Since for a row i we need to fill in each column $0 \leq j \leq i$, we still need to handle the $j = i$ case. This one is slightly different; because the lowest-ℓ unmerged table is $T[i]$, we explicitly can't merge $T[i]$ and have no restrictions beyond that (since our restriction says nothing about the availability of tables with ℓ values greater than $T[j].\ell$). So, we should just pick the optimal unrestricted solution to the ($T[0]$ to $T[i-1]$) subproblem, which is just the max of the restricted solutions (we add nothing to its value since we're not adding a new merge to the solution):

$$M[i][j] = M[i-1][\underset{0 \leq k < i}{\text{argmax}}\{M[i-1][k].gains\}]$$

Using this piecewise recurrence relation, we can compute all of M. Once this is done, our actual solution is just the cell in the bottom row (ie, the "subproblem" using the entire input) which has the max `gains` value.

4 An example

Suppose we have that

$$K = 50, T = [t_1, t_2, t_3, t_4]$$

$$t_1 = \{\ell = 9, \texttt{count} = 40\}$$

$$t_2 = \{\ell = 6, \texttt{count} = 10\}$$

$$t_3 = \{\ell = 5, \texttt{count} = 20\}$$

$$t_4 = \{\ell = 3, \texttt{count} = 30\}$$

Representing values we haven't computed yet as dots and using a concise representation of a matrix cell, after setting our base case value we have:

$$M = \begin{bmatrix} \{\texttt{g} : 0, \texttt{m} : []\} & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

This makes sense so far; using only one table we can't have any merges. Now let's fill in the next row:

$$M = \begin{bmatrix} \{\texttt{g} : 0, \texttt{m} : []\} & \cdot & \cdot & \cdot \\ \{\texttt{g} : 20, \texttt{m} : [(t_2, t_1)]\} & \{\texttt{g} : 0, \texttt{m} : []\} & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

$M[1][0]$ is just us taking the cell directly above it (ie, the best configuration possible such that the lowest-ℓ table available is t_1) and applying a merge of t_2 into the best merge target available; ie, t_1. ($\texttt{gains} = M[0][0].\texttt{gains} + K - t_2.\texttt{count} \cdot (t_1.\ell - t_2.\ell) = 0 + 50 - 10 \cdot (9 - 6) = 20$).

$M[1][1]$ requires us to *not* merge t_2 into anything, so we just take the best result from above. Let's do the next row now:

$$M = \begin{bmatrix} \{\texttt{g} : 0, \texttt{m} : []\} & \cdot & \cdot & \cdot \\ \{\texttt{g} : 20, \texttt{m} : [(t_2, t_1)]\} & \{\texttt{g} : 0, \texttt{m} : []\} & \cdot & \cdot \\ \{\texttt{g} : -10, \texttt{m} : [(t_2, t_1), (t_3, t_1)]\} & \{\texttt{g} : 30, \texttt{m} : [(t_3, t_2)\} & \{\texttt{g} : 20, \texttt{m} : [(t_2, t_1)]\} & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

This is just more of the same; note though that there's nothing preventing us from going into negative gains. Certainly it's never an optimal thing to do overall (and in fact, this algorithm does do more work than it needs to by continuing to process columns which go into negatives, but I'm not convinced removing this extra work would actually result in a better runtime complexity, so I've opted to keep the algorithm simpler here), but it is an optimal thing to do within the restriction imposed by this column (the obviously better alternative to doing a negative-gain merge is to just not do the merge, but that would violate the column restriction by making our lower-ℓ table available for merging into).

$$M = \begin{bmatrix} \{\texttt{g} : 0, \texttt{m} : []\} & \cdot & \cdot & \cdot \\ \{\texttt{g} : 20, \texttt{m} : [(t_2, t_1)]\} & \{\texttt{g} : 0, \texttt{m} : []\} & \cdot & \cdot \\ \{\texttt{g} : -10, \texttt{m} : [(t_2, t_1), (t_3, t_1)]\} & \{\texttt{g} : 30, \texttt{m} : [(t_3, t_2)\} & \{\texttt{g} : 20, \texttt{m} : [(t_2, t_1)]\} & \cdot \\ \{\texttt{g} : -130, \texttt{m} : [(t_2, t_1), (t_3, t_1), (t_4, t_1)]\} & \{\texttt{g} : -10, \texttt{m} : [(t_3, t_2), (t_4, t_2)\} & \{\texttt{g} : 10, \texttt{m} : [(t_2, t_1), (t_4, t_3)]\} & \{\texttt{g} : 30, \texttt{m} : [(t_3, t_2)\} \end{bmatrix}$$

Now that we've finished our algorithm, we can see that the best option from the last row gains us 30 bytes and involves a single merge of t_3 into t_2. I also think it's cool that looking at the final table you can see the algorithm weighing the pros and cons of merging t_2 into t_1; making this merge gains us 20 bytes on its own and is the clear winner when looking at the second row, but the third row shows us that the gains from leaving t_2 unmerged outweigh the gains of merging it into t_1 by 10 bytes.