Radboud University Nijmegen

Complexity Pet Problem

Robbert van der Gugten Steven Reitsma

June 19, 2014

1 Sum solver

Initially we devised an algorithm with a powerful heuristic that we believed to be able to solve the problem in $\mathcal{O}(nm)$ time. The algorithm is based on summing the amount of compatibilities in each row and column in the compatibility matrix. This leaves us with $n+m$ sums. We then greedily assign pets to children based on the values in the sums, every iteration choosing the indexes where the sums are lowest. This means we assign 'picky' children/pets first, then moving on to the more accepting children and pets. The algorithm works perfectly for the examples, however, it is not guaranteed to be optimal if the heuristic cannot make an accurate prediction. Consider the example compatibility matrix below:

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{1.1}$$

The sums of the rows and columns are all 2. The algorithm now picks the first pet (column) to assign first. It then checks at what row-indices there is a 1 in the compatibility matrix; these indices are 0 and 1 for the example above. Finally it selects the index with the lowest row-score (both are equal in this example, so the algorithm just picks the first row) and thus assigns Pet 0 to Child 0. We now remove row 0 and column 0 from our sum list, which would result in the following matrix:

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \tag{1.2}$$

Since we do not update the sums during the algorithm, the next steps also assign pets and children to each other at random, because all sums are equal and the algorithm thus does not have a preference to select a certain column or row earlier. In we finish this example, we will have the following assignments:

Pet 0 to Child 0
Pet 3 to Child 1
Pet 1 to Child 2

Now we cannot assign Pet 2 to Child 3, which would be the only pet left for Child 3. Thus we can prove by example that this algorithm is not optimal, since a better assignment would be:

Pet 0 to Child 0
Pet 3 to Child 1
Pet 2 to Child 2
Pet 1 to Child 3

If we were to recompute the sums between each assignment, the algorithm would work on this counter example. However, this would increase the complexity from $\mathcal{O}(nm)$ to $\mathcal{O}(nm \times nm) = \mathcal{O}(n^2m^2) \in \mathcal{O}(n^4)$.

2 The ScoreSolver algorithm

In this algorithm a score is assigned for every value in the compatibility matrix using bottom-up processing. The scores are stored in a new matrix. The score is determined by the following equation:

$$s_{i,j} = max(s_{i+1} \setminus s_{i+1,j}) + c_{i,j} \tag{2.1}$$

where c is the compatibility matrix and s is the score matrix. For the bottom row, the first part of the equation is ignored. Thus the scores of the bottom row are equal to the compatibility matrix.

So the score is determined by the previous row without considering the jth element, plus the score in the compatibility matrix. After the score matrix is filled the pets are assigned by choosing the highest scores first, taking into account the pets that already have been chosen. If there are two or more same scores then the chosen pet is determined by the value that is lost when choosing that pet. The one with the lowest loss value is chosen. All these loss values are determined by the remaining scores in that row, for example, the loss value of element (3,3) is: score(3,3) + score(2,3) + score(1,3) + score(0,3). All these values are stored in a matrix for quick access.

2.1 Example

Consider the following compatibility matrix:

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \tag{2.2}$$

Using equation 2.1 this results in the following score matrix:

$$\begin{bmatrix} 5 & 5 & 4 & 4 \\ 3 & 3 & 4 & 4 \\ 2 & 3 & 2 & 2 \\ 2 & 1 & 2 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \tag{2.3}$$

To determine the loss values, we compute the cumulative values in a bottom-up fashion:

$$\begin{bmatrix} 12 & 12 & 13 & 12 \\ 7 & 7 & 9 & 8 \\ 4 & 4 & 5 & 4 \\ 2 & 1 & 3 & 2 \\ 0 & 0 & 1 & 1 \end{bmatrix} \tag{2.4}$$

Processing this will result in the following assignment:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.5}$$

Note that there are still two choices for the first assignment $(0,0)$, even when taking the loss value into account. However, it does not matter which one is chosen first, both assignments will give an optimal result.
$(1,3)$ is assigned because they both have the highest score, but the value of $(1,2)$ is 9 and $(1,3)$ is 8, the lowest is chosen which is $(1,3)$.

2.2 Complexity

The time complexity of this algorithm is in $\mathcal{O}(mn)$ and the space complexity is also in $\mathcal{O}(mn)$. We used two dynamic programming techniques to attain this low complexity: memoization and bottom-up processing, both learned during the course.

2.3 Optimality

We believe the algorithm is optimal since for every step in the process the optimal option is chosen. The algorithm is not greedy because we use bottom-up processing to calculate the scores and loss values beforehand. We were unable to find a counter example to disprove our hypothesis. Further work should be done to draw a conclusion.