Objects flowing through a video

Sebastian Damrich
Heidelberg University
sebastian.damrich@gmx.de

Lukas Bold
Heidelberg University
lukasbold92@gmail.com

Abstract

The abstract paragraph should be indented ½ inch (3 picas) onboth the left- and right-hand margins. Use 10 point type, with a vertical spacing (leading) of 11 points. The word **Abstract** must be centered, bold, and in point size 12. Two line spaces precede the abstract. The abstract must be limited to one paragraph.

1 Introduction

In this report, we discuss strategies for tracking an object in a video given the initial position of the object. For training more positions of the object are available. The main idea of all our approaches is to treat the movement of the object as the flow in a graph that is build from connecting the pixels of one frame of the video with the pixels of the next frame. The pixels through which there is flow shall indicate the position of the object. The flow is computed by a variant of the minimal-cost flow problem (MCFP) whose parameters are the output of a neural network. We follow two main trains of thought, which were both suggested by Prof. Hamprecht. In the first one, we examine ways of differentiating through the optimisation problem of the MCFP. We explain why ideas along the lines of [AK17] are not suitable for our problem and introduce workarounds for special cases of the MCFP. The second train of thought considers a special version of the MCFP, namely a version of optimal transport. Here, we do not try to differentiate through the optimisation problem itself, but differentiate through the steps of Sinkhorn's algorithm which solves the entropy regularised problem.

2 Related Work

3 Differentiating through a MCFP

Out first strategy is to model the movement of the object in the video as a flow which is minimal with respect to some costs. Then we try to differentiate through the MCFP to improve the costs.

3.1 Min-Cost-Flow programs

Given a directed graph $G = (V, E)$ with edge weights c, edge capacities κ and vertex supply b, the MCFP is the problem of finding a flow of minimal cost along the edges of the graph, which respects the capacity constraints and manages to match up the vertices with positive and negative flow supply. This can be formulated as a linear program:

$$\begin{aligned} &\min_f \sum_{e\in E} c(e)\, f(e) \text{ s.t.} \\ \forall v \in V : &\sum_{(v,w)\in E} f(v,w) = \sum_{(u,v)\in E} f(u,v) + b(v) \\ \forall e \in E : &0 \le f(e) \le \kappa(e) \end{aligned} \tag{1}$$

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

where the first condition ensures that flow is conserved and the supply and demand at each vertex are met while the second condition guarantees the capacity constraints. In the following, we will often write c, b, κ and f as vectors of length $|E|$.

3.2 MCFP for object tracking

We construct the directed graph on which we want to compute a min-cost-flow as follows: The vertices are given by the pixels of the video in addition of an auxiliary node t, which serves as a sink. There are edges from a pixel to a square area of pixels below it in the next frame (pixels close to the borders of the frame have less outgoing edges). We call the length of the sides of this square window size. It should be an odd number. All the pixels in the last frame have an edge to the sink. The flow supply is one for every pixel in the first frame that belongs to the object. This encodes that we know the position of the object in the first frame. All other pixels have a flow supply of zero apart from t which has a flow demand matching the cumulative flow supply.
The costs and capacities of the the graph are the output of a neural networks, which takes the video as input.
Given a min-cost-flow for the above described graph, we predict the location of the object in every frame by the pixels through which there is a positive flow. We can also associate a loss l to a flow. This is done subtracting the the amount of flow that goes through the object in the last frame from the amount of flow that misses the object in the last frame. As a result, we could capture the change of the loss by the flow by the gradient:

$$\left(\frac{\partial \ell}{\partial f}\right)_{(v,w)} = \begin{cases} 1 - 2 \cdot \mathbb{1}(\text{the pixel to } v \text{ is part of the object}) & \text{if } w = t \\ 0 & \text{else} \end{cases} \tag{2}$$

In order to train the neural network, we need to know $\frac{dl}{dc}$ and $\frac{dl}{d\kappa}$. Given $\frac{dl}{df}$ as above and using the chain rule, the key is to find $\frac{df}{dc}$ and $\frac{df}{d\kappa}$. In other words, we need to find out how the optimal flow, the solution to the linear program 1, changes with the problem parameters c and κ. For this task we explored the ideas of [AK17]. If the position of the object in more than the first and last frame is known at training time, one could include them into the loss function. If the position of the object in the last frame is know at testing time, one could omit the sink and uniformly distribute the flow demand over the object's position in the last frame. In this chapter, we focused on the situation where the location of the object in the first and last frame is known at training time but only the initial position is known at test time.

4 Special cases of the MCFP

We saw in the above section, that the approach of [AK17] seems not well suited for our problem. Instead of differentiating through the optimisation problem to obtain a gradient for training the neural networks, we therefore use a solver to solve the MCFP problem and then update the parameters of the problem in a way that reduces the loss at least heuristically at the solution. Those updated parameters then serve as targets for the neural network In our analysis, we specialised the general MCFP to two corner cases. This reduces the computation time and highlights different features of costs and capacities.

4.1 Shortest Paths

We explored the situation without upper capacity constraints in some detail. Without upper capacity constrains, the MCFP becomes the problem of finding the shortest paths from the pixels that constitute the object in the first frame to the sink. One advantage of this approach is that there are fast solvers for Shortest Paths problems, such as the Dijkstra or the Bellman-Ford algorithm. [1] Those paths that do not go through object in the last frame, have lost the object at some point. Therefore, we penalise them by increasing the cost of the edges along such a path by a penalty parameter. One the other hand, paths that go through the object in the last frame are rewarded. The costs along their edges is reduced by the penalty parameter.
The shortest paths algorithm gives the shortest paths from the background of the first picture to the

[1] Due to the special directed structure of our graph, finding shortest paths can be computed by the Dijkstra algorithm (and thus a version of message passing) even when costs are negative.

sink for free. In order to increase the amount of information passed to the neural network, we also update the costs along the paths from the background to the sink in the same fashion as before, but in opposite direction: Paths from the background that pass through the background of the last frame are rewarded and those that pass through the object in the last frame are penalised.
One issue with this way of updating costs is that if a path that is penalised and one that is rewarded have a common end (they start in the object and the background respectively, but merge at some point), the rewards and penalties on the common end cancel. The possibility of the merging of paths itself could also be an issue. If an edge has very small cost, it might induce several paths to use this edge and hence to merge. Moreover, without upper bounds on the capacities there is no possibility for two paths to separate once they have merged. This would be particularly problematic when the object increases in size during the video. In general, it might happen that many paths merge at some point and that as a result the last frame is only passed at very few pixels. This would yield degenerated predictions of the objects location. Fortunately, this over-merging does not seem to be an issue, as we will see in section 7.2.

DIFFERENTIATE THROUGH MESSAGE PASSING

4.2 Maximum Flow

The other extreme case of a MCFP would be to focus entirely on the capacities rather than the costs. By setting the costs to zero for all edges, the cost-minimisation problem becomes ill-defined. In order to dodge this problem, one could replace the cost-minimisation problem with a flow maximisation problem. Instead of defining a fixed flow supply for the nodes in the first frame and the sink, one could allow the pixels that constitute the object in the first frame to send as much flow as possible and the sink to take in as much flow as possible. For all other pixels flow-in equals flow-out still applies. The resulting linear program would look like this

$$\begin{aligned} &\max_f \sum_{(v,t)\in E} f(v,t) \text{ s.t.} \\ \forall v \in V \setminus (V' \cup \{t\}) \quad &\sum_{(v,w)\in E} f(v,w) = \sum_{(u,v)\in E} f(u,v) \\ &\forall e \in E : 0 \leq f(e) \leq \kappa(e) \end{aligned} \tag{3}$$

where V' are the pixels in the first frame that constitute the object. One advantage of such a simplification of the MCFP is that there are several fast algorithms for solving a maximum flow problem, for instance the Ford-Fulkerson (or Edmund-Karp) algorithm or the Pre-Flow-Push algorithm. So as above, one could let a neural network predict capacities given the video, then find a maximum flow through the resulting network and finally update the capacities in a meaningful way to provide an update direction for the neural network. Similar to the case of shortest paths, one way to update the capacities is to increase the capacities along a path of positive flow to the object in the last frame and to decrease the capacities (bearing in mind the lower bound of zero capacity) along a path of positive flow to a background pixel in the last frame. A theoretical appeal of this strategy is that it nicely matches up with the notion of augmenting paths that are used to determine a maximum flow with the Ford-Fulkerson algorithm.
Another advantage of using capacities rather than costs is that maximum flow behaves more continuously and piecewise even differentiably with regard to the capacities in contrast to the case of a change in the costs. Changing the capacity of an edge on which the flow is not maximal infinitesimally, does not change the flow at all. Decreasing the capacity of an edge on which the flow is maximal, reduces the flow on that edge with a derivative of one. Increasing the capacity on such an edge either increases the flow on that edge with derivative one (if the network allows more flow to enter and leave the edge) or it does not change the flow on that edge at all. The effect on other edges can be manifold, due to the non uniqueness of a maximal flow on some networks, however there always are maximal flows that change piecewise differentiably on all edges with respect to the edges.
Normalising the loss from section 3.2 by the size of the flow and pretending that changing the capacity of an edge with maximal flow always results in a derivative of one yields a crude approximation to a derivative $\frac{\partial \ell}{\partial \kappa}$:

$$\left(\frac{\partial \ell}{\partial \kappa}\right)_{(v,w)} = \begin{cases} \frac{1-2\cdot \mathbb{1}(w \text{ is part of the object})}{\sum_{(u,t)} f(u,t)} & \text{if } w \text{ is in last frame and } f(v,w) = \kappa(v,w) \\ 0 & \text{else} \end{cases}$$

However, this is more or less just updating the capacities of edges leading towards the last frame depending on whether they lead to the object or not. This latter approach however seems to resemble more an object detection scheme than an object tracking one.
An upside of using capacities is that in contrast to using shorest paths capacities can handle a magnifying object well.

5 Optimal transport via the Sinkhorn algorithm

In this section we explore another variant of the MCFP for object tracking. We model the object's location as a probability distribution over the set of pixels of a frame. Given the first and last position of the object we want to predict its position in the intermediate frames. With distributions d^1 and d^2 for the position of the object in two consecutive frames and a cost matrix C for transporting flow between any pixel in one frame to any pixel in the other frame, one can compute the Earth Mover's distance between them, which is the following linear problem

$$\begin{aligned} &\min_P \langle P, C\rangle \text{ s.t.} \\ &P^T \cdot \mathbb{1} = d^1, P \cdot \mathbb{1} = d^2 \\ &\forall i,j : P_{ij} \geq 0 \end{aligned} \tag{4}$$

where $\langle P, C\rangle = \sum_{i,j} P_{ij}C_{ij}$. This can be seen as a transport problem and as such as a MCFP without upper capacity constraints. The linear problem is infeasable to solve for large dimensional distributions as in our case. In order to speed up computation, [Cut13] adds an entropy regularisation term of the form $-\frac{1}{\lambda}h(P) := \frac{1}{\lambda}\sum_{i,j} P_{ij}\log(P_{ij})$ to the objective function. The solution of this entropy regularised version of the problem has a special form, due to which it can quickly be computed using Sinkhorn's algorithm ([Sin67] and [Cut13] Lemma 2).
In our situation, we consider one distribution for each frame of the video, $d^1, \dots, d^N$, of which d^1 and d^N are given as uniform distributions over the object's location in the first and last frame. The others are unknown. As before, we let a neural network compute the costs from the video. Let C^t denote the cost for transporting flow between the pixels of frame t and $t+1$. Candidates for the object's position in all frames, could be those distributions $d^2, \dots, d^{N-1}$ for which the optimal transport is minimal:

$$\begin{aligned} &\min_{\{P^t\}_{t=1}^{N-1}, \{d^t\}_{t=2}^{N-1}} \sum_{t=1}^{N-1} \langle P^t, C^t\rangle \text{ s.t.} \\ &\forall t : P^{t,T} \cdot \mathbb{1} = d^t, P^t \cdot \mathbb{1} = d^{t+1} \\ &\forall i,j,t : P^t_{ij} \geq 0. \end{aligned} \tag{5}$$

Note that the d^t's are automatically distributions by the above constraints and the fact that d^1 is. This problem is equivalent to first computing the lowest cost, C^*, for transporting flow from a pixel in the first frame to one in the last frame by finding shortest paths through the graph given by connecting all pixels of one frame with all pixels in the next frame. Using C^*, we are reduced to the classic EMD problem 4. From its solutions we can then construct all the P^t's and d^t's by following the shortest paths computed C^*. As in [Cut13], we could speed up the computation time of this equivalent problem by adding an entropic regularisation term. But this might be insufficient to train the neural network well as there is still the discrete computation of shortest paths involved. As discussed above shortest paths behave "jumpy" in the costs and are thus hard to differentiate through. We can try to avoid this by regularising even further. Instead of one global entropic term we could add entropic regularisers for each P^t in 5. To ease notation, we also substract the constant $\frac{N-1}{\lambda} = \frac{1}{\lambda}\sum_{t,i,j} P^t_{ij}$.
From resulting Lagrangian, L, we can deduce:

$$L\left(\{P^t, \alpha^t, \beta^t\}_{t=1}^{N-1}\right) = \sum_t \langle P^t, C^t\rangle + \frac{1}{\lambda}h(P^t) + \alpha^{tT}(P^t \cdot \mathbb{1} - d^{t+1}) + \beta^{tT}(P^{tT} \cdot \mathbb{1} - d^t) \tag{6}$$

$$P^t_{ij} = \exp\left(-\frac{\alpha^t_i}{\lambda}\right)\exp\left(-\frac{M^t_{ij}}{\lambda}\right)\exp\left(-\frac{\beta^t_j}{\lambda}\right) \tag{7}$$

In other words, the transport matrices again have the form

$$P^t = D(u^t) \cdot K^t \cdot D(v^t)$$

for $K^t = \exp\left(-\frac{C^t}{\lambda}\right), u^t = \exp\left(-\frac{\alpha^t}{\lambda}\right), v^t = \exp\left(-\frac{\beta^t}{\lambda}\right)$ elementwise. If the distributions d^t were all fixed, we could again use the simple iterations of the Sinkhorn algorithm [Sin67] to find the P^t. They come from the fact that the row and column sums of P^t must be d^{t+1} and d^t, respectively, and are:

$$u^t = d^{t+1} \oslash K^t v^t \text{ and } v^t = d^t \oslash K^{tT} u^t, \tag{8}$$

where $\oslash$ ($\otimes$) means elementwise division (multiplication). Since $d^2, \ldots, d^{T-1}$ are also optimisation variables, we have to update them too, for instance via:

$$d^{t+1} = u^t \otimes K^t v^t \text{ and } d^t = v^t \otimes K^{tT} u^t. \tag{9}$$

Alternating between the iterations in 8 and 9 leads to many but very simple updates. Despite this not being a true instance of the Sinkhorn algorithm, we hope that the process converges fast. After some stopping criterion is reached, we could compute a cross entropy loss between the $\{d^t\}_{t=2}^{T-1}$ and distributions coming from the true position of the object. This loss could be differentiated by the d^t's and then by unrolling the iterations of 8 and 9 also by the costs, so that we can train the neural network. Once the network is trained we can use this approach to track an object in a video, given its position in the first and last frame. We could also make this approach more similar to the one before by only fixing d^1 and computing the loss just using d^N.

6 Implementation

We have implemented the shortest paths approach.[2] For the neural network we drew inspiration from the architecture of [LZL+17]. The network is a convolutional network with a shallow U-form. A ReLu layer follows each convolutional layer. As described in [LZL+17], at first two 2D convolutions are applied. We use a kernel size equal to the window size of the graph. The 2D convolutions increase the channel number from 3 over 5 to 10. Padding is used to keep the original dimensions. The result is recorded, let us call it X and a copy is 2D max pooled to decrease the dimensions of the frames by a factor of 2. Then, the video is 3D convoluted to introduce time dependency. The 3D convolutions keep the channel number constants. First, we use a kernel of dimensions (3, window size, window size). So, the convolution sees a square area of length window size in three consecutive frames. It should be noted, that this corresponds to information being convoluted that comes from an area of side length twice the window size as we have pooled before. We apply padding to keep the dimensions. Afterwards, we use a kernel of dimensions (2, window size, window size) with padding to keep the latter two dimensions. As a result, we reduce the time dimension by one. Then we upsample trilinearly keeping the time dimension but restoring width and height of the frames to the original values, let us call the result Y. Before adding the 2D information X and the 3D information Y, we convolute X in the time dimension with a kernel of size 2 to match up dimensions. In the last step, we apply a 2D convolution layer with a kernel of window size, which gives out window size^2 many channels. Here we do not apply a ReLu layer. We interpret the channels of the output as the costs for edges going out of a pixel.
The graph is built explicitly using the graph_tool package. Since we build a large graph, we chose this performant package. [3] Finally, we use a Mean Squared Error between the computed and the updated costs as loss for the neural network.

DOCUMENT THE GITHUB

7 Experiments

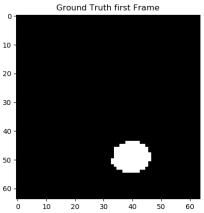
7.1 Datasets etc

7.2 Results

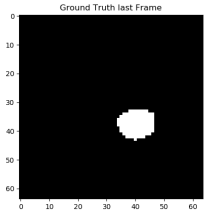
We have tested the algorithm for window sizes 3, 5, 9 and 13 and training iterations for the network of 10, 50 and 200 on the first 5 frames of the cropped and shrunk video "ball1". The resulting predictions are shown in figure **??**. The ball moves upwards between the first and last frame. The main result is

[2] `https://github.com/sdamrich/FlowTracker`

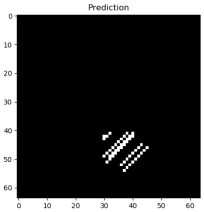
[3] `https://graph-tool.skewed.de/performance`



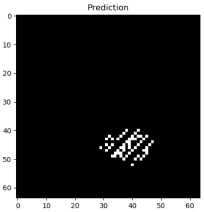
(a) GT first frame



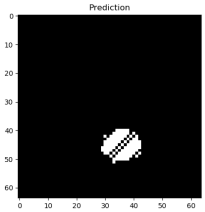
(b) GT last frame



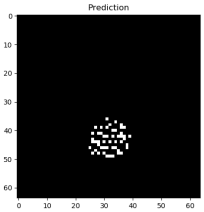
(c) w = 3, iter = 10



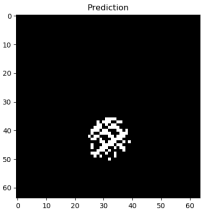
(d) w = 3, iter = 50



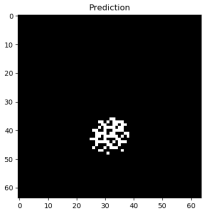
(e) w = 3, iter = 200



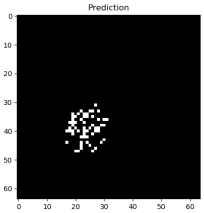
(f) w = 5, iter = 10



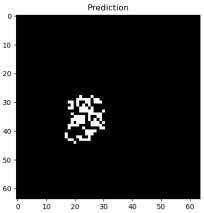
(g) w = 5, iter = 50



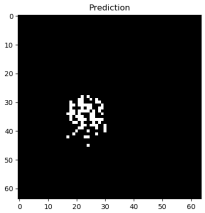
(h) w = 5, iter = 200



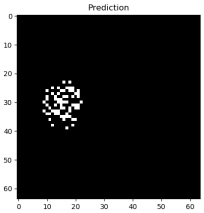
(i) w = 9, iter = 10



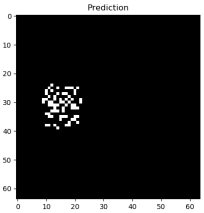
(j) w = 9, iter = 50



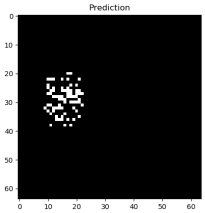
(k) w = 9, iter = 200



(l) w = 13, iter = 10



(m) w = 13, iter = 50



(n) w = 13, iter = 200

Figure 1: Ground truth and predictions for $w \in \{3, 5, 9, 13\}$ and iter $\in \{10, 50, 200\}$

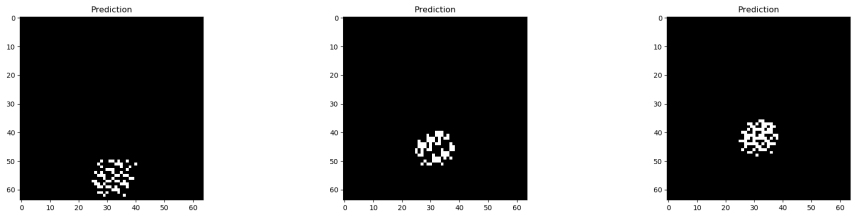


Figure 2: Predictions of three different trainings with window size 5 and 200 training iterations.

that the algorithm tends to preserve the shape of the ball, but does not follow its movement, even though it was trained and tested on the same video. Therefore, we refrained from scaling up our experiments to several training and test videos. The ball moves upwards so fast that it can only be properly tracked with window sizes 9 and 13.

We see that the anticipated over-merging problem from section 4.1 does not appear. The predictions in the last frame, though misplaced are rather dense and of the correct shape. We believe that this is because we use a convolutional network for computing the costs. When the network is told that some cost should be changed, it has to update its convolution filters which are applied in the same way to every area of the image. This way, the network avoids to only apply changes locally. As a result, no edges reduce their cost too much compared to the rest that there is the danger of over-merging of paths. In fact, the predictions are sparser for fewer iterations when the network only had little time to train. With more training the predictions become dense and the shape of the prediction looks more like a ball.

However, even with 200 iterations the random initialisation of the network seems to matter as we can see in figure **??**, where the predictions for three different training periods with a window size of 5 and 200 training iterations are depicted. The ball is predicted to be at very different locations.

The MSE between the computed and updated costs does not change much in the course of the training, which is why we do not report it in detail. This is not surprising, as we always update every shortest path to the sink. So the only possibility for the loss to decrease is if the updates along two paths cancel on a mutual part. In particular, even a perfect flow that exactly tracks the object, would result in a loss of similar magnitude as one that entirely misses the object. One could avoid this by scaling the update by a factor which decreases with the overlap of the flow in the last frame and the ground truth.

One can see easily that the overlap of the predictions and the ground truth given by $\frac{\text{area of intersection of prediction and GT}}{\text{joint area of GT and prediction}}$ is rather small and in fact for most of our experiments the prediction misses the ground truth entirely. The exact values can be found in table 1. The better overlaps are achieved for small window size and high iterations. We believe that unfortunately, this is not because the smaller window size are generally better (they are even too small to theoretically track the object correctly) but since for small window sizes, the prediction has no chance to escape too much to the left.

A common effect seems to be that instead of just moving upwards, the algorithm predicts the ball to also go to the left. This effect increases with the window size and with the number of iterations. We do not know why the object gets moved to the left by the algorithm and suspect an undetected programming error on our side.

It makes sense that the overall movement increases with the window size, simply because a larger window size allows for larger movement from frame to frame. With a large enough window size

Table 1: Overlap of prediction and ground truth

		Iterations		
		10	50	200
Window size	3	0.227	0.302	0.411
	5	0.094	0.110	0.075
	9	0.000	0.000	0.000
	13	0.000	0.000	0.000

and thus large freedom of movement, the algorithms predicts the ball even higher than it is in the ground truth ((j)-(n)). This might be because our training iterations do not converge. In most cases the prediction is as high and as far left as the window size allows. Only in the case of window size 13, the prediction could theoretically be even higher. The fact that the movement also slightly increases with the iterations might show that there is some burning in period.

8 Conclusion

There are several possible ways of using Min-Cost Flows for object tracking in videos. Such an optimisation problem looks at the whole video at once and does not propagate the position of the object iteratively through the video. This way information about the location of the object in different frames can be used in a unified manner. Issues with using MCFPs for object tracking is the large resulting problem size, which can make directly soling the linear problem or even differentiating through it prohibitively expensive. Another problem is the non-continuous change of an optimal flow with respect to the costs. Both the long computation time and the poor differentiability could potentially be mitigated by regularising the linear problem. Thus, it would be interesting to pursue and implement the ideas of section REF.

References

[AK17] Brandon Amos and J. Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. *CoRR*, abs/1703.00443, 2017.

[Cut13] Marco Cuturi. Sinkhorn distances: Lightspeed computation of optimal transport. In *Advances in neural information processing systems*, pages 2292–2300, 2013.

[LZL+17] Kisuk Lee, Jonathan Zung, Peter Li, Viren Jain, and H. Sebastian Seung. Superhuman accuracy on the SNEMI3D connectomics challenge. *CoRR*, abs/1706.00120, 2017.

[Sin67] Richard Sinkhorn. Diagonal equivalence to matrices with prescribed row and column sums. *The American Mathematical Monthly*, 74(4):402–405, 1967.