CSC148: A Closer Look at the Grouping Algorithms for Assignment 1

Suppose we have the following set of students in a course. For simplicity, we're only listing their IDs and names:

ID: 1 Name: Sophia	ID: 2 Name: Jaisie	ID: 3 Name: Mario	ID: 4 Name: Diane	ID: 5 Name: Misha

ID: 6 Name: Christine	ID: 7 Name: Alex	ID: 8 Name: Amin	ID: 9 Name: Nidhi

Each of the Grouper classes (AlphaGrouper, GreedyGrouper, and SimulatedAnnealingGrouper) will group these students in different ways, described in their docstrings and below.

AlphaGrouper

The AlphaGrouper class will group students alphabetically. Suppose we're forming groups of size 2. The groups formed should then be:

ID: 7 Name: Alex	ID: 8 Name: Amin

ID: 6 Name: Christine	ID: 4 Name: Diane

ID: 2 Name: Jaisie	ID: 3 Name: Mario

ID: 5 Name: Misha	ID: 9 Name: Nidhi

ID: 1 Name: Sophia

Where the students are arranged alphabetically, and then split into groups accordingly.

GreedyGrouper

The GreedyGrouper class forms groups in a "greedy" manner: We first get a list of students ordered by ID.

ID: 1 Name: Sophia	ID: 2 Name: Jaisie	ID: 3 Name: Mario	ID: 4 Name: Diane	ID: 5 Name: Misha

ID: 6 Name: Christine	ID: 7 Name: Alex	ID: 8 Name: Amin	ID: 9 Name: Nidhi

The first student who is not part of a group is placed into the first group. In this case, Sophia is in the first group.

ID: 1 Name: Sophia

ID: 2 Name: Jaisie	ID: 3 Name: Mario	ID: 4 Name: Diane	ID: 5 Name: Misha

ID: 6 Name: Christine	ID: 7 Name: Alex	ID: 8 Name: Amin	ID: 9 Name: Nidhi

Afterwards, the student that would increase this group's score the most (or reduce it the least) will be added to that group, breaking ties by ID.

Suppose we get the following changes to the first group's score for each student:

ID: 1 Name: Sophia

ID: 2 Name: Jaisie **Score +0**	ID: 3 Name: Mario **Score +2**	ID: 4 Name: Diane **Score -1**	ID: 5 Name: Misha **Score +5**

ID: 6 Name: Christine **Score +4**	ID: 7 Name: Alex **Score -2**	ID: 8 Name: Amin **Score +3**	ID: 9 Name: Nidhi **Score +5**

Then Misha would be added to the first group:

ID: 1 Name: Sophia	ID: 5 Name: Misha

ID: 2 Name: Jaisie	ID: 3 Name: Mario	ID: 4 Name: Diane

ID: 6 Name: Christine	ID: 7 Name: Alex	ID: 8 Name: Amin	ID: 9 Name: Nidhi

We would repeat the previous step until the group reaches our intended size, finding the change in scores for the remaining students and adding the student that would increase the score the most (or decrease it the least) into the group.

Suppose we're using groups of size 2. Then this first group is finished, and we repeat our steps with the next student who has yet to be put into a group, Jaisie:

ID: 1 Name: Sophia	ID: 5 Name: Misha

ID: 2 Name: Jaisie

ID: 3 Name: Mario	ID: 4 Name: Diane	ID: 6 Name: Christine	ID: 7 Name: Alex	ID: 8 Name: Amin	ID: 9 Name: Nidhi

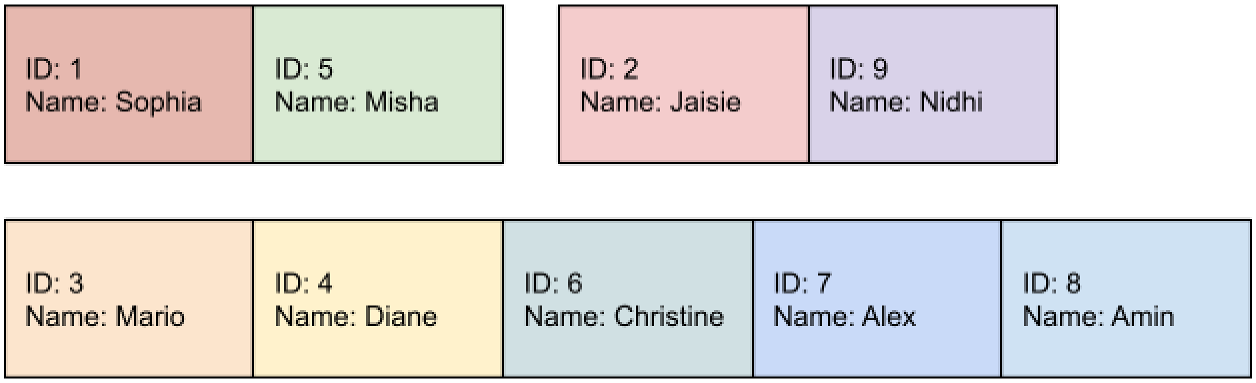
And suppose each student would result in the following changes to the score of Jaisie's group when added:

ID: 1 Name: Sophia	ID: 5 Name: Misha

ID: 2 Name: Jaisie

ID: 3 Name: Mario **Score -15**	ID: 4 Name: Diane **Score -3**	ID: 6 Name: Christine **Score -6**	ID: 7 Name: Alex **Score -3**	ID: 8 Name: Amin **Score -4**	ID: 9 Name: Nidhi **Score -1**

Then Nidhi would be added into Jaisie's group, being the one who reduces it the least:



These steps would repeat until we form enough groups.

This approach is called “greedy” because it always looks for the best step it can take – the one student who would be best to add to the current group – but it does so without concern for the longer-term consequences. For instance, maybe adding that student to a later group would make for a much better overall result. But it’s too late. We have short-sightedly added them to this group, and the algorithm doesn’t reconsider decisions made. This makes it fast but not optimal.

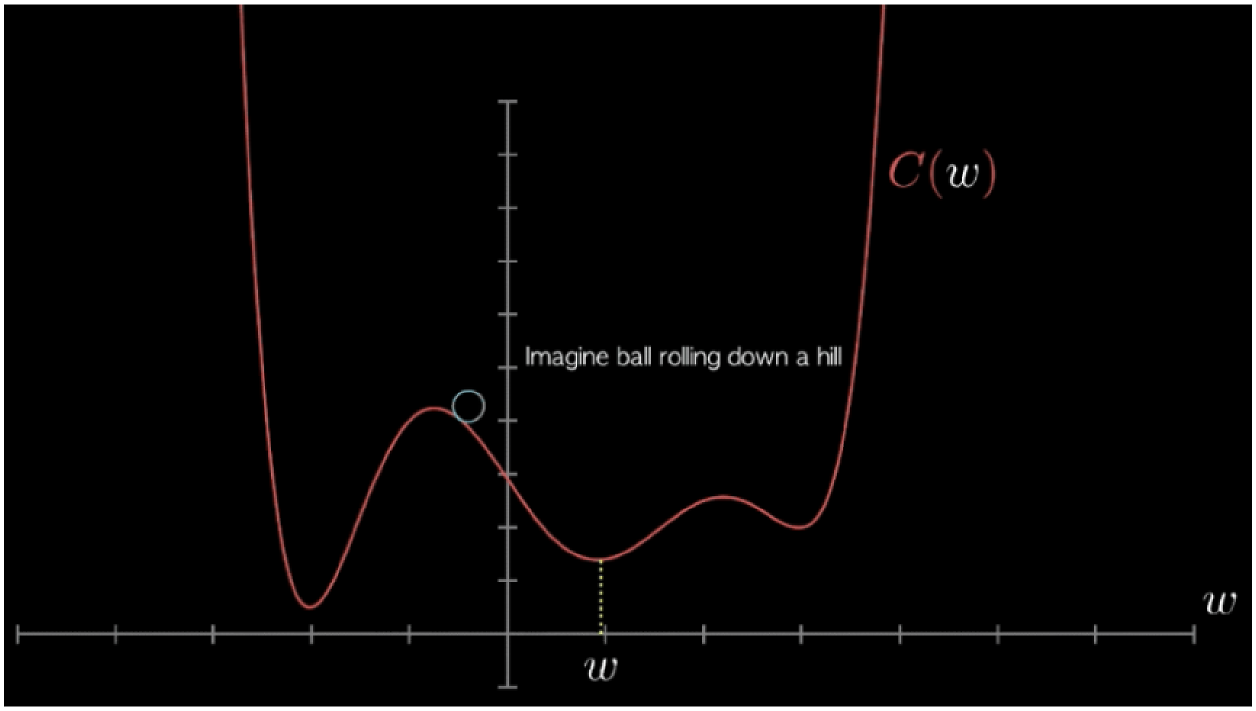
SimulatedAnnealingGrouper

The SimulatedAnnealingGrouper begins by making a random grouping. It might be a good one or a terrible one, but this doesn’t matter because the grouper goes on to try improving the grouping using the following strategy: generate an alternative grouping by making a small, randomly-chosen modification to the grouping we have, and then decide whether to adopt the new grouping or keep the old one. By doing this over and over, it is likely to find a reasonably good grouping.

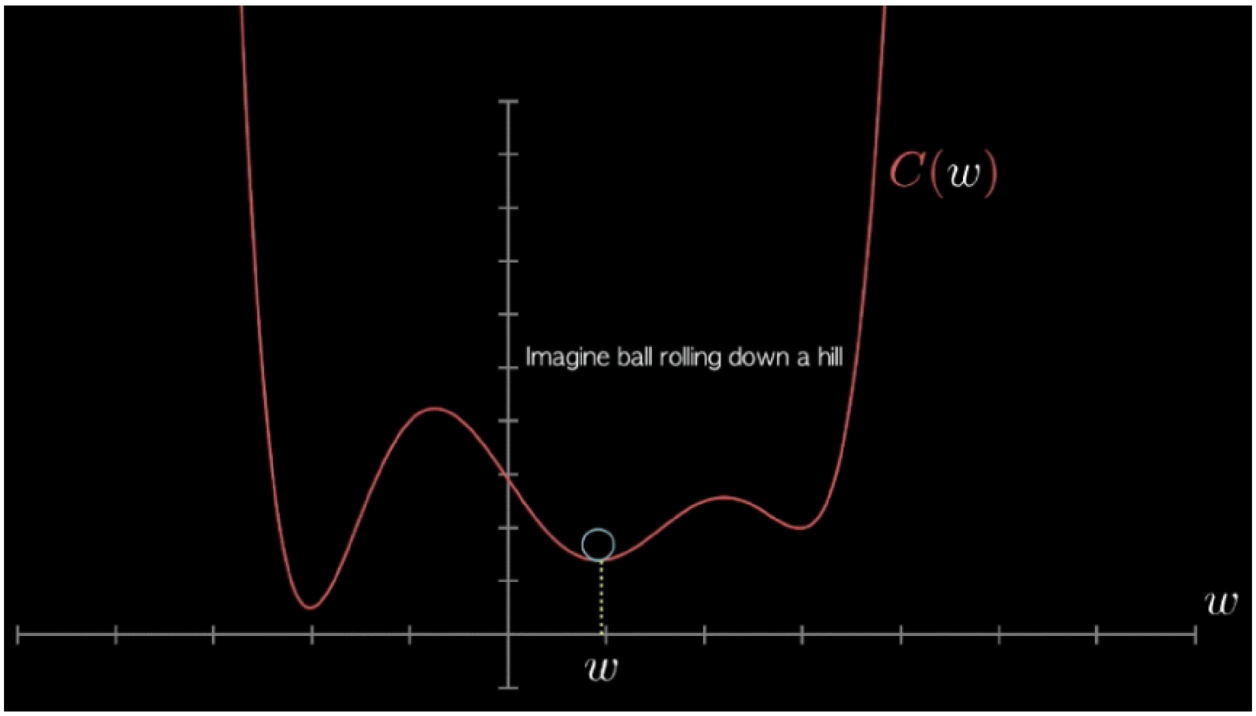
With this algorithm, we search for a good answer by looking at alternatives that are just a *little* different from (not too “far away” from) what we currently have. By only looking at “nearby” solutions rather than all possible solutions, a lot of time is saved. And often the final result is a pretty good solution. Since this algorithm doesn’t consider every possible solution – in this case, every possible way to divide students up into groups – we can never know if we found the best solution overall.

Analogy to finding the minimum of a function

Consider the problem of finding the global minimum (the “lowest point”) of this curve (source, with animation), starting from the location of the ball.



One simple strategy is to move in the best of the two directions (this is a “greedy” choice, similar to choosing the best student to add in the GreedyGrouper algorithm above). In this case, the ball would move to the right, since values to the right are smaller and we are looking for the lowest point.



Using this same strategy of moving in the “best” direction, the ball would end up in the dip in the curve *w* above. Unfortunately, this is not the global minimum, but any move from here, whether left or right, goes up; so we are stuck here with this greedy strategy.

A Way to do Better

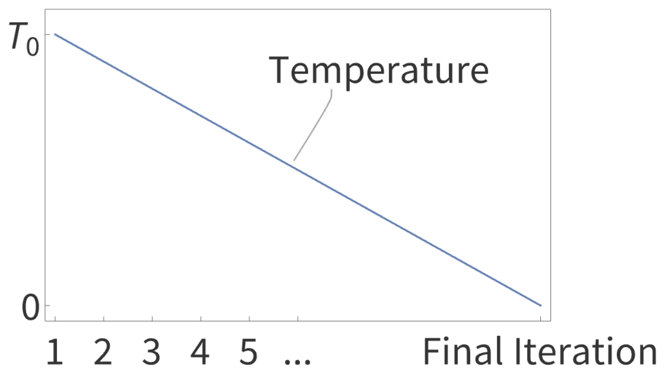
“Simulated annealing” is an algorithm that tries to avoid this pitfall. It sometimes accepts solutions that have a worse score (i.e., it sometimes moves up the hill instead of down). By doing this, it gives us a chance to search the possibilities more widely, raising the probability that we find the optimal answer. And it still avoids having to look at every possible solution.

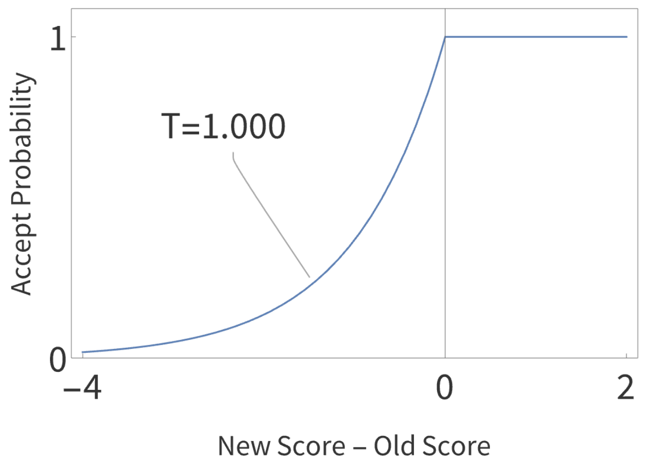
The term “annealing”, which means “cool slowly,” refers to the idea that the algorithm starts with a “high temperature”. In our metaphor of a rolling ball, we think of the ball as having a lot of energy, so it can jump around widely and explore much of the solution space. As the algorithm progresses, the temperature gets cooler and we tend to only consider the nearby choices.

In <SimulatedAnnealingGrouper.make_grouping>, at each step

1. A new random grouping is generated that is “nearby” to the existing grouping, i.e., not very different from the existing grouping.
2. The algorithm “decides” if it should accept the new grouping or continue with the previous one. The current “temperature” is used to determine how open we will be to accepting a new grouping with a lower score.
3. Repeat steps 1 and 2 for a fixed number of iterations.

We have provided a more detailed version of this algorithm for you in the docstring of the <SimulatedAnnealingGrouper.make_grouping> method.





Two critical components of the simulated annealing algorithm are the ***Temperature*** and the ***Acceptance Probability Function.***

Temperature

The temperature is a positive number that starts at some value T_0 and decreases over the iterations until it reaches zero. There are many sensible ways to define the temperature, but in this assignment, we will use the most standard technique: the temperature starts at T_0 and decreases linearly towards zero. An example temperature function, whereT_0 = 1, is shown above).

Here is how to compute the temperature. Let N be the total number of iterations we have chosen for the algorithm, and let i denote the current iteration number of the algorithm (it will go from 0 to N-1). The temperature on the ith iteration is defined by:

$$T_i = T_0 \times \left(1 - \frac{i}{N-1}\right)$$

You will need to write code to update to the temperature with each iteration as part of implementing <SimulatedAnnealingGrouper.make_grouping>.

The Acceptance Probability Function

The acceptance probability function is a mathematical function that determines the probability with which the new grouping should be accepted. It does this based on two values:

1. The difference between the score of the new grouping and the score of the current grouping: If this is positive, the new grouping is better than the current one, and the probability of accepting it should be high; but if the difference is negative, the new grouping is worse and the probability of accepting it should be lower.
2. The temperature: If the temperature is high, we will be willing to accept a new grouping that decreases our score some of the time; the probability of doing so will not be zero. But as the temperature goes down, we become less and less willing to do so. We have a high temperature when we are early in our number of iterations; that is, we've just started looking. We have a low temperature when we are almost done looking.

We chose to accept a new grouping with a probability of 1 if the score is better, and otherwise with a probability equal to the exponential of the improvement over the temperature. The following equation summarizes this:

$$\text{ProbAccept}(\text{temp}, \text{new score - old score}) = \begin{cases} 1 & \text{new score} - \text{old score} \geq 0 \\ \exp\left(\frac{\text{new score}-\text{old score}}{\text{temp}}\right) & \text{otherwise} \end{cases}$$

Note that in the special case where temp=0 we use the equation

$$\text{ProbAccept}(0,\ \text{new score - old score}) = \begin{cases} 1 & \text{new score} - \text{old score} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

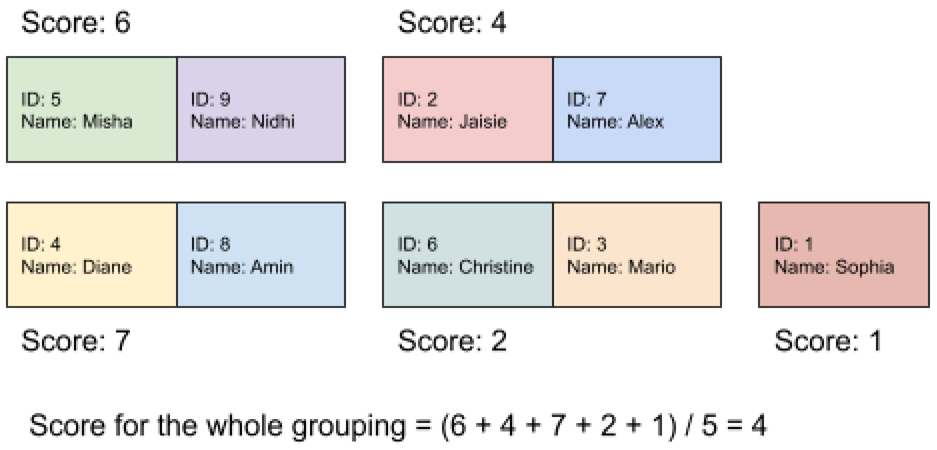
The Python `accept` Function

In `grouper.py`, we have provided for you a helper function called `accept` that uses the temperature which you pass to it, the acceptance probability function above (which we have implemented), and a randomly generated number, to decide whether or not to accept a new grouping. We are providing this explanation here to help you understand what that code is doing.

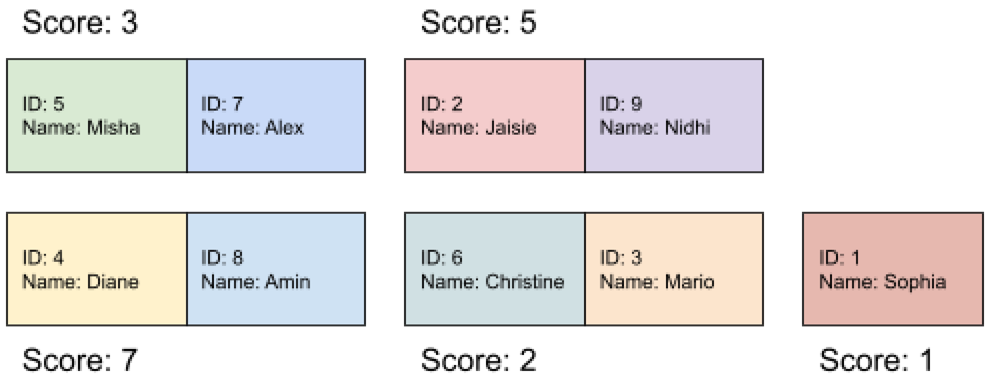
Example

In this example, the initial temperature T_0 is 1, group size is 2, and the total number of iterations is 100.

We start with a random grouping and compute the grouping score (the average score of all the group scores). For this example, we have made up scores, but in the actual assignment, scores are based on the survey provided.



Our method to create new groupings will be to select two groups at random and swap a random member from each of them. For example, suppose we swap Nidhi and Alex, and suppose this leads to a grouping with a slightly lower score:



Total Score = (3 + 5 + 7 + 2 + 1) / 5 = 3.6

If the new score were better, we'd accept it. But since this score is lower than the previous one, we need to decide whether or not to accept the new grouping. In the assignment, you will call the <accept> function.

That function will compute the acceptance threshold using the current temperature *T_0 = 1*:

$$\exp((3.6-4)/1) = 0.67032$$

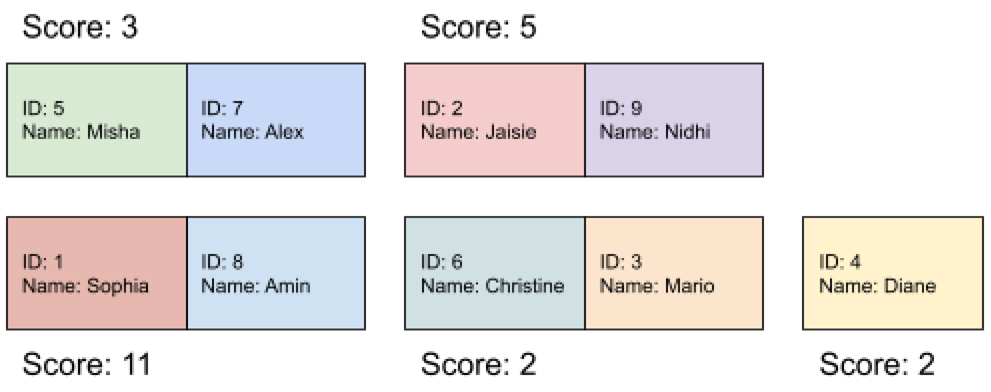
This means the new grouping will be accepted with a probability of 67.032%, even though it is "worse" than the previous one. This is achieved by choosing a random number x between 0 and 1 and accepting the new grouping if and only if $x \leq 0.67032$. *(As mentioned, function* `accept` *does this computation for you, you do not need to implement it yourself.)*

For instance if we generated the random number $x = 0.457327$ we would accept the new grouping, with a score of 3.6, as the current solution and use it in our next iteration. On the other hand, if the generated random number was $x = 0.811294$, we would *not* accept the new group and would revert to the previous solution.

Regardless of whether we accept it or not, we must update our temperature parameter for the next iteration

$$T_1 = 1 \times \left(1 - \frac{1}{100-1}\right) = 0.989899$$

Assume we accept the grouping above, even though it leads to a lower score. Now, let's say in the next iteration, Diane and Sophia are swapped. In this case, suppose this leads to a grouping with a higher score:



Total Score = (3 + 5 + 11 + 2 + 2) / 5 = 4.6

This grouping is guaranteed to be accepted because it produces a higher score. We would then update the temperature again.

This process of generating another random grouping, deciding whether to accept it, and updating the temperature, will repeat until we have reached the number of iterations specified (in this example, 100 iterations).

An important note about random seeds

The SimulatedAnnealingGrouper requires the use of random numbers in several places. In order for us to be able to test that your code is correct, you must generate random numbers in a special way that is not truly random, known as pseudorandom number generation.

In the SimulatedAnnealingGrouper we will perform all random operations using a specific "seed" in order to be sure that our tests run exactly the same as your code.

The two functions that require seeding are `random_swap`, and `accept`. To "seed" one of these functions, just send it a particular value for its `seed` parameter. **You must always seed these functions using the iteration number of the algorithm.**