**Problem Statement:** How do we best arrange stadium seating to maximize both profit for the stadium (in the form of most seats sold) as well as safety for the viewers?

**Constraints:**

* The stadium seating area can be any triangle or quadrilateral
* Seating blocks can be of any size, but in current state can only be p x q rectangles and completely vertical line arrangements are not considered
* The seating buffer between blocks is 2 seats on all sides, and 1 seat diagonally
* Ingress and egress are not of concern for now, but an aisle buffer exists as a preliminary solution to this problem

Here’s an example seating block where m = 10 and n = 6:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

In this seating block we can fit exactly two 16 seat cubes and still comply with our seat distancing buffer of 2 seats. This would look something like this:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

**Implementation:**

This seat-placing algorithm will be implemented in python on a local machine (laptop).

Input: A “seating chart” document in txt format with 0s representing each seat in the rectangular seating configuration. A 10x6 seating area like the one shown above would look like this:

0000000000

0000000000

0000000000

0000000000

0000000000

0000000000

Output: Another “seating chart” document in txt format but this time with 0s representing seats that must remain empty due to the necessary seating buffer and 1s representing sellable tickets. Using the same model as above, the output file would look like this:

1111001111

1111001111

1111001111

1111001111

0000000000

0000000000

Conveniently, the sum of this grid is also the number of seats the stadium can sell, in this case, 32. It would be very easy to make the first line of the file be the number of seats sold.

Architecture:

A screenshot of a cell phone

Description automatically generated

Run-through Explanation:

The program consists of several files, but two carry the principle execution: engine.py and run.py. Run.py imports the run function from engine.py, so all running should occur from run.py.

*engine.py:* The implementation of engine.py rests on several structures, simplifications, and helper functions. One of the first simplifications is that seating arrangements are stored in table format (called seatingTable or seatingArea), which takes the form of a list of a list. Each inner list represents a row of the seatingTable. Each element in the inner lists is a seat. For example, the seatingTable [[0, 0, 0], [1, 0, 1]] would look something like this:

000

101

Recall that 1s represent filled seats and 0s empty seats, so the total number of seats filled in this arrangement is 2. Notice that this simplification allows for easy calculation of the number of seats filled (it’s the sum of the table).

Seats are filled in so-called seating blocks, which are of a specified size “s”. Recalling that more efficient seating arrangements are accomplished with larger block sizes, the algorithm prioritizes a certain seating block size first, denoted as the variable “s”. Later on, the function squeezeBlocksIn does a post-processing sweep of all sizes n s.t. 0 < n < s.

The algorithm operates by generating all of the possible seating block arrangements that can be filled, and then trying each to see if it’s safe given the various buffers specified (namely xBuffer, yBuffer, diagonalBuffer, aisleBuffer, and aisleDirection). The algorithm iterates over every seat in seatingArea, and if it’s safe to put a block there, it places it. It alternates randomly between placing blocks of size s and blocks of size 1, 2, 4, or 8 to place more desirably sized seating blocks. Since blocks are of different shapes and sizes, the algorithm chooses amongst these options randomly. Once the entire arrangement is maximally filled by the random algorithm and the best arrangement is picked, it is passed onto the squeezeBlocksIn function to try to fill in blocks of smaller sizes than s.

Though this brute force algorithm is inefficient and could certainly be improved using optimization to place blocks in a less random way, its power comes in the number of seatingTables it generates via the automateWhile function. For the number of reps specified in this function, automateWhile generates that many seatingTables. Once a list of all these arrangements is generated (recalling that each is filled by both fillRectangle and squeezeBlocksIn), it’s passed onto the pickBestArrangement function to choose the best arrangement. Currently, this is decided by the goodnessFunction, which weighs the degreeFilled (number of seats filled in a certain arrangement compared to the maximum number of seats that were able to be filled amongst all) and proportionNotS (the percentage of seats in the seatingTable that are not of size s). Recalling that more efficient arrangements are accomplished with larger blocks, this balances our two desires: efficient seating, and desirable (or smaller) seating blocks.

In order to keep track of all seating blocks in the seatingTable, each seatingTable has its own dictionary called sizeTrackDict, keepTrackDict, or masterDict depending on the context. Each key in this dictionary is an integer size s, and each value is the integer number of blocks of that size in the seatingTable. For example the dictionary {3: 2} would indicate that there are 2 blocks of size 3 in a seatingTable.

All variables like block size, buffer specifications, and aisle direction can be changed in lines 457-464.

*run.py:* The run file is far simpler: it imports all of the backend implementation functions calls from the engine file in the form of the run function. The function goes through each file in the input folder and executes the run function on it, passing an output file into the output folder for each seating arrangement. If changes are made to propNotS (namely the desired seating block size s) and goodnessFunction in engine.py, it’s important these changes are mirrored in the same function calls in run.py.

Certainly, there are other files that aren’t important to program execution but have been used in the past or may assist with future implementations. These files include convergencedata.py and polyomino.py.

*convergencedata.py:* This file was used in the past to calculate and record both runtime and goodness scores of past implementations of the seating algorithm. It is now obsolete as changes made in engine.py were not passed into convergencedata.py. Regardless, all data was stored in convergence.log and showed that runtime was linear (O(n)) compared to the number of seatingTables generated (reps). Additionally, it (problematically) showed that as reps increased, goodness score did not change significantly. A better implementation of the seating algorithm should ensure that (1) runtime does not increase exponentially for runtime’s sake and (2) goodness score increases as more runtime is invested, otherwise time would be wasted.

*polyomino.py:* This file starts with a disclaimer. Most of the code was borrowed from an online repository operated by Félix Saparelli. Nonetheless, I made several of my own additions and edits to adapt it to my needs.

I turned to polyominoes in an attempt to allow the constraint on seating blocks to be extended to allow for non-rectangular blocks. The code in this file generates all polyominoes of a given size n, and then filters them to only select ones we consider “good”, which I defined to be those with at least 2 seats adjacent in each horizontal slice, meaning no individual would be seated with nobody beside them, obviously with the exception of seating blocks of size 1. However, this still proved to be too many seating block shapes, especially for larger seating block sizes. Therefore, it was left out of implementation until improvements could be made.

This project was undertaken by Matt Jogodnik (Duke c/o 2022) under the guidance of Anthony Volpe (Quantworks, Inc.) Any questions can be directed to mij5@duke.edu.