The International Mathematical Modeling Challenge
Aboard! Boarding and Disembarking a Plane
Контрольный номер команды:
2022RU17

2022
Всероссийский конкурс по математическому моделированию

1 Аннотация

Данная статья посвящена изучению разных способов посадки людей в салон самолета. Рассмотрено 10 различных методов посадки в узкофюзеляжный самолет (имеющий только один проход), для каждого из которых создана агентная модель клеточного типа, позволяющая анализировать среднее время посадки всех людей на свои места. Также рассмотрен случай неполной загрузки самолета и других данных, которые могут меняться при переходе от абстрактного представления в реальную жизнь. Далее проведен анализ полученных данных, в процессе чего рассматривалось применимость каждого из методов в реальных условиях. Получен фаворитный метод, который, по нашему мнению, стоит применять при посадке. Также создана универсальная модель, описывающая самолеты с любым количеством проходов, которая дает представление в первую очередь о самолетах с двумя проходами и самолетах вида Flying Wing, хоть на первый взгляд его и сложно представить в гражданских аэропортах. Описаны допущения, используемые в программе и проанализировано влияние процента людей, не соблюдающих правила, на итоговый выбор метода.

Ключевые слова: посадка в самолет, эффективные способы посадки/высадки, моделирование процесса посадки/высадки, метод Штеффана

Содержание

2 Письмо руководителю авиакомпании

Уважаемый руководитель авиакомпании,

Наша команда провела исследование, результаты которого могут Вас заинтересовать, как организатора авиасообщения.

Суть исследования заключалась в том, чтобы определить самый эффективный метод посадки людей на борт самолета, что является актуальной темой в сфере вашей деятельности. Мы рассмотрели 10 разных методов посадки и создали математическую модель, которая способна выявить из предложенных вариантов самый эффективный. Причем результат зависит от заданной жизненной ситуации, то есть от вида самолета, процента загруженности, количества ручной клади и остальных критериев, которые ваша компания задает своим клиентам перед полетом. Другими словами, наша модель способна для каждого из производимых вашей компанией рейсов, владея информацией о количестве проданных билетов и других основополагающих факторов, определять рациональный метод посадки.

В нашем исследовании мы провели сравнения только нескольких из возможных ситуаций (полная загрузка, загрузка на меньший процент людей и размещение пассажиров, среди которых есть нарушители предписанного метода посадки), но уже они позволили нам судить о выигрышных стратегиях посадки пассажиров и представить эти результаты - демонстрацию работы нашей модели.

Таким образом, по результатам проведенной работы наша команда считает, что стоит обратить пристальное внимание и по возможности использовать посадку “пирамида”. Эта посадка показала лучший результат в своей категории и при этом ее реализация возможна в реальной жизни (уже есть компании, которые используют этот метод посадки, экономя свое время).

Хоть основную и решающую роль играет итоговое время размещения людей, комфорт пассажиров также является очень важным, хотя сложно измеримым фактором, который также стоит учитывать. При этом могут возникнуть “проблемы друзей”, то есть произойдет разделение на время посадки группы знакомых людей, летящих и сидящих вместе. Мы предлагаем вам в случае, если группа не может быть разделена (например, родитель с ребенком), проводить посадку всей компании с той посадочной группой, наименьшая из которых встречается среди билетов рассматриваемой компании, причем речь идет лишь о тех группах людей, которые будут сидеть на одном ряду, а в остальных же случаях все равно советуем разделять людей, даже если они летят вместе. Таким образом, будет достигаться наилучший метод посадки, удовлетворяющий все желания пассажиров.

Мы надеемся, что наше письмо заинтересовало вас и позволило лучше узнать о методе нашей работы. Будем рады сотрудничеству и с удовольствием ответим на все возникшие вопросы.

3 Введение

3.1 Актуальность проблемы

Специалисты постоянно исследуют процессы, которые происходят вокруг нас. К примеру, они пытаются оценить пользу и вред от электромобилей или ищут методы борьбы с пробками. Но заторы могут создавать не только машины, но и люди. Например, при посадке в самолёт.
Авиакомпании заинтересованы в том, чтобы свести к минимуму время, отведенное на посадку и высадку пассажиров, поскольку любая минута простоя воздушного судна стоит больших денег. Для авиакомпаний, инфраструктуры аэропорта и пассажиров сокращение времени посадки означает снижение эксплуатационных расходов и улучшение качества обслуживания.

3.2 Формулировка задачи

Основной задачей, стоящей перед командой, стали разработка и анализ разных методов посадок с помощью созданной нами математической модели. Также анализ влияния таких факторов как загруженность салона, доля людей, не соблюдающих предписанные правила посадки, и количество ручной клади у каждого из пассажиров на способ посадки. Причем такие исследования должны были быть проведены для самолетов трех типов: узкофюзеляжного, с двумя входами и двумя выходами и самолета типа “Flying Wing”.

3.3 Методы исследования

Для рассмотрения этой задачи нами были разработаны агентные модели двух типов. Первый тип основывается на принципе клеточного автомата, а именно место, занимаемое объектом (пассажиром, креслом и т. д.), считается равным одной клетке на клетчатом поле. Такой способ позволяет получить примерные результаты, но при этом не учитывает различных габаритов пассажиров, разную скорость их движения и другие моменты, которые упускаются из виду, но при этом могут сыграть значительную роль в реальных условиях.
Второй же тип основывается на принципе сеточного автомата (объекты занимают несколько клеток). Такой тип позволяет учитывать габариты людей, их скорости, а также то, что в некоторых случаях в проходе может помещаться два человека, например, если один агент складывает ручную кладь, стоя в проходе боком, а второй пытается пройти дальше, то он может это сделать, не дожидаясь того, как первый закончит, но с меньшей по сравнению с изначальной скоростью. Так, данный подход позволяет оценивать методы посадки способом более приближенным к реальности.

4 Исследование

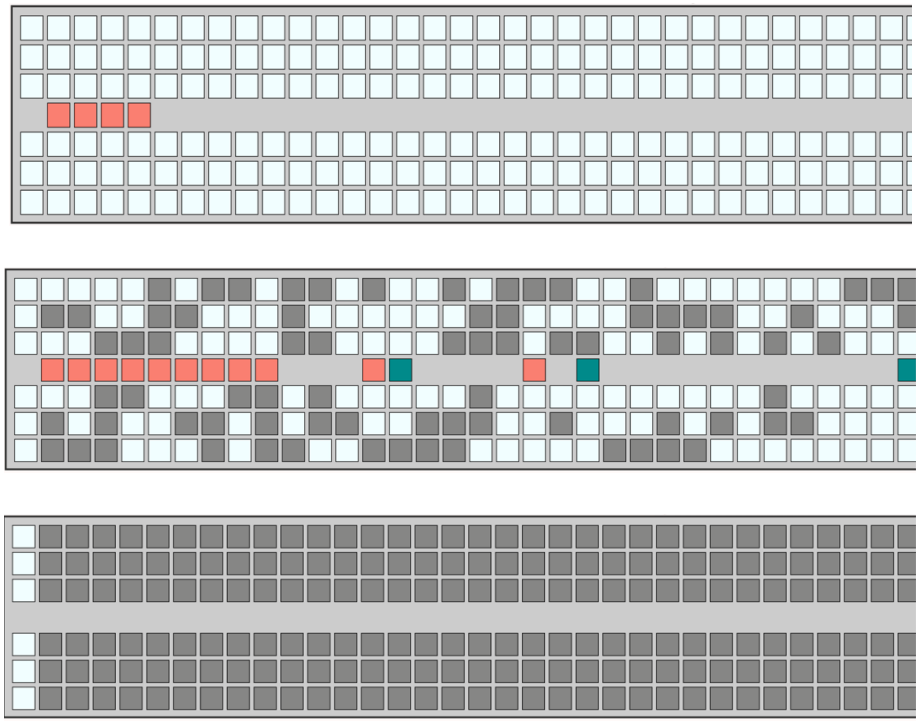
4.1 Простая модель

При написании программы клеточного типа считается, что люди идут по проходу, не обгоняя друг друга, и занимают места в порядке, соответствующем рассматриваемому методу.

Реализуется это так:

1. Имеется список, хранящий в себе координаты всех мест в том порядке, в каком впоследствии будут заниматься места в самолете. То есть если рассматривается хаотичный метод, то и в списке координаты мест находятся в хаотичном порядке. Если же рассматриваемый метод подразумевает деления людей на группы, то в списке координаты упорядочены в соответствии с принадлежностью к группе, причем внутри одной группы координаты имеют хаотичный порядок. В то же время в этом списке количество людей соответствует рассматриваемому проценту загруженности самолета.
2. Считается, что за единицу времени объект двигается на одну ячейку. При этом в проходе может находиться только один человек, и если впереди идущий останавливается, чтобы сложить ручную кладь и занять свое место, то все люди за ним останавливаются.
3. Также программа учитывает случай, если место заблокировано (то есть пассажир хочет занять место у окна, хотя место у прохода уже занято). В таком случае ко времени ожидания посадки этого человека добавляется время, необходимое для освобождения прохода к месту, то есть время, за которое уже сидящие люди встанут, а потом сядут обратно. Соответственно на это же время увеличивается время ожидания всех людей, идущих за данным пассажиром.
4. На каждом шаге мы создаем массив, отражающий данное положение системы и с помощью библиотеки tkinter можем визуализировать происходящее

Отметим также, что все рассматриваемые методы посадки для узкофюзеляжного самолета описываются в одном коде, то есть основная часть для всех методов одинаковая, а меняется лишь способ задания массива из п.1. Также в программе можно менять загруженность самолета, время за которое человек кладет ручную кладь (с помощью чего оценивается влияние количества ручной клади на итоговый ответ), а также влияние людей, которые не соблюдают правила (в упорядоченном по группам массиве (см. п.1) переставляются местами несколько значений, количество которых соответствует выбранному количеству нарушителей). Таким образом, наша программа описывает все рассматриваемые методы и легко поддается базовому анализу чувствительности методов посадки.



4.2 Результаты

Спроектировав эту модель, мы начали ее исследование на тех рассадках, которые были предложены в условии задания. Всего для каждой рассадки было произведено порядка 10 тысяч запусков программы, результатом работы которой был файл, содержащий информацию о доле севших пассажиров в каждый момент времени. По этим данным был построен график *N(t)* процент севших пассажиров от времени, который представлен на *диаграмме 1* — она соответствует максимуму загрузки.

Ввиду недетерминированности алгоритма посадки получалось так, что при разных запусках получалось разное время для посадки. Таким образом, необходимо было получить некоторое среднее значение по временам посадок и сказать, что за это среднее время мы можем рассадить всех пассажиров

$$Tk = \frac{1}{10000} \sum_{i=1}^{10000} Ti$$

Tk - усредненное время посадки, Ti - время посадки по результату запуска

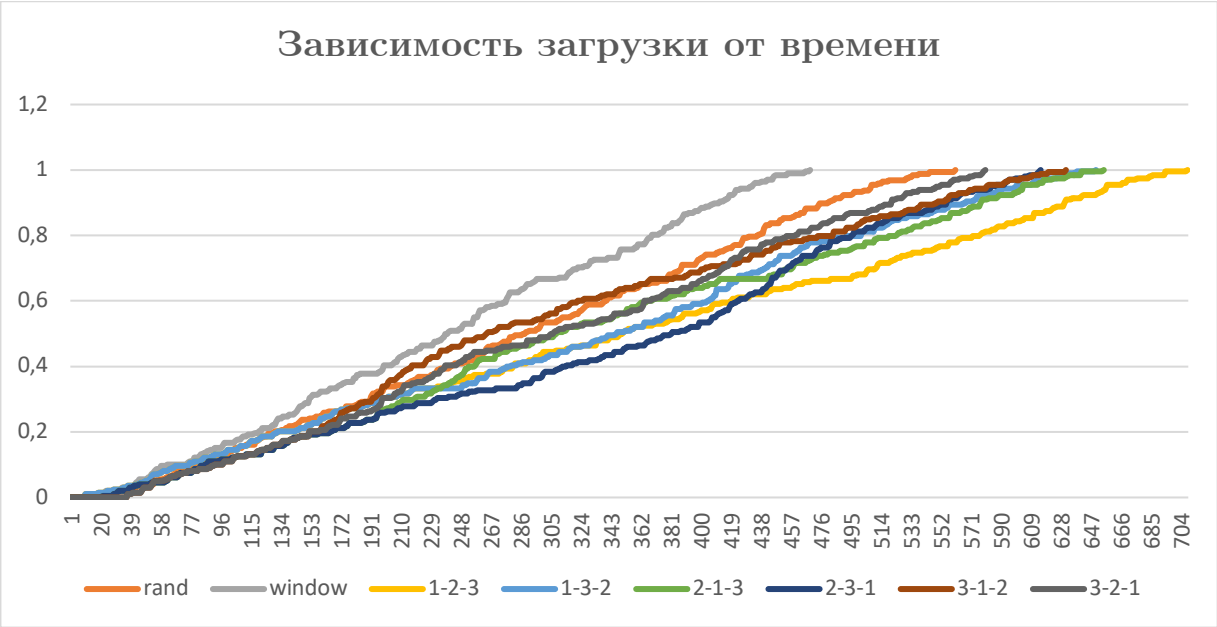


Диаграмма 1

Комментарий к диаграмме: на горизонтальной оси отложено время в секундах, на вертикальной - доля севших пассажиров. Кривая rand означает, что рассадка производилась в случайном порядке, window - сначала рассаживаются пассажиры у иллюминаторов, затем те, кто сидят посередине, после чего люди, сидящие в проходе; остальные 6 посадок — это всевозможные комбинации из посадок по группам 1–11 ряды, 12–22 ряды и 23–33 ряды

Как нетрудно понять, лучший результат показала рассадка, при которой сперва садятся пассажиры у окна, затем те, чьи места в середине, а потом пассажиры, сидящие с краю. Это имеет простое объяснение: когда сначала размещаются пассажиры у окна, пассажирам, места которых расположены по середины, не нужно дополнительно ждать, пока освободится проход до их места.
Причем разница по сравнению с произвольным методом, который использует большая часть авиакомпаний, оказывается порядка 13.8% (с результатом 1016 и 876 секунд при произвольном выборе времени укладывания багажа из диапазона от 10 до 12 секунд).
Интересно проследить за тем, что происходит при изменении среднего количества ручной клади на человека и процента загрузки.
Как оказывается, несмотря на варьирование этих параметров, опыт показывает, что результат остается примерно тем же. Поэтому мы не будет приводить полученные графики, а лишь скажем о их результате.

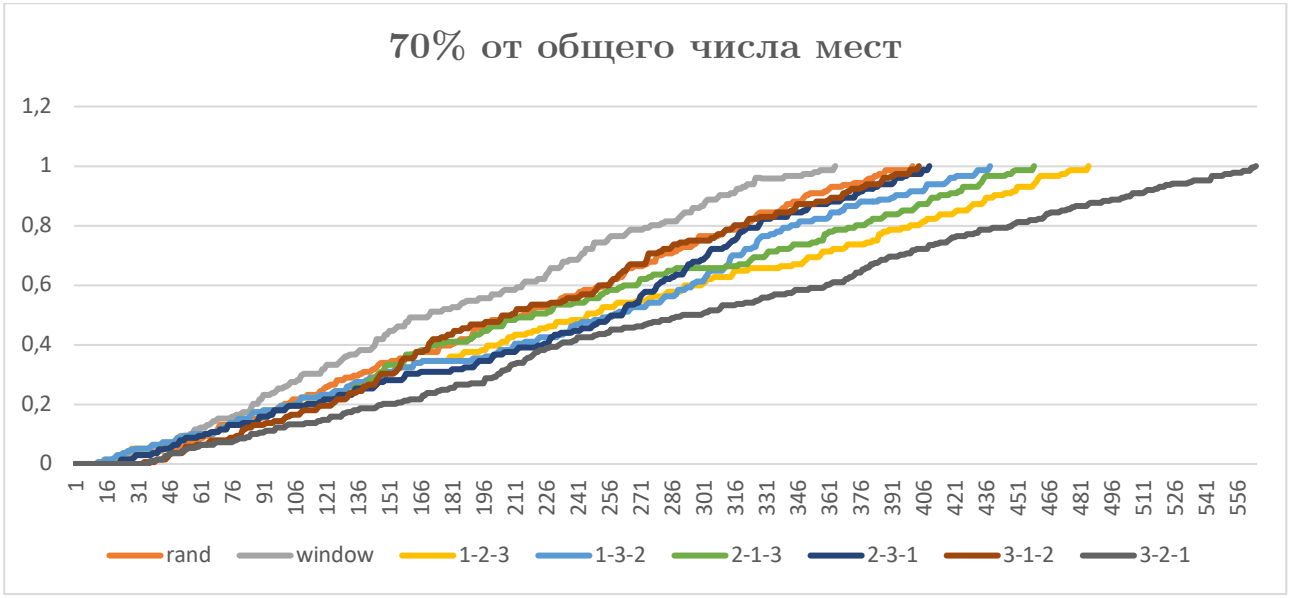


Диаграмма 2

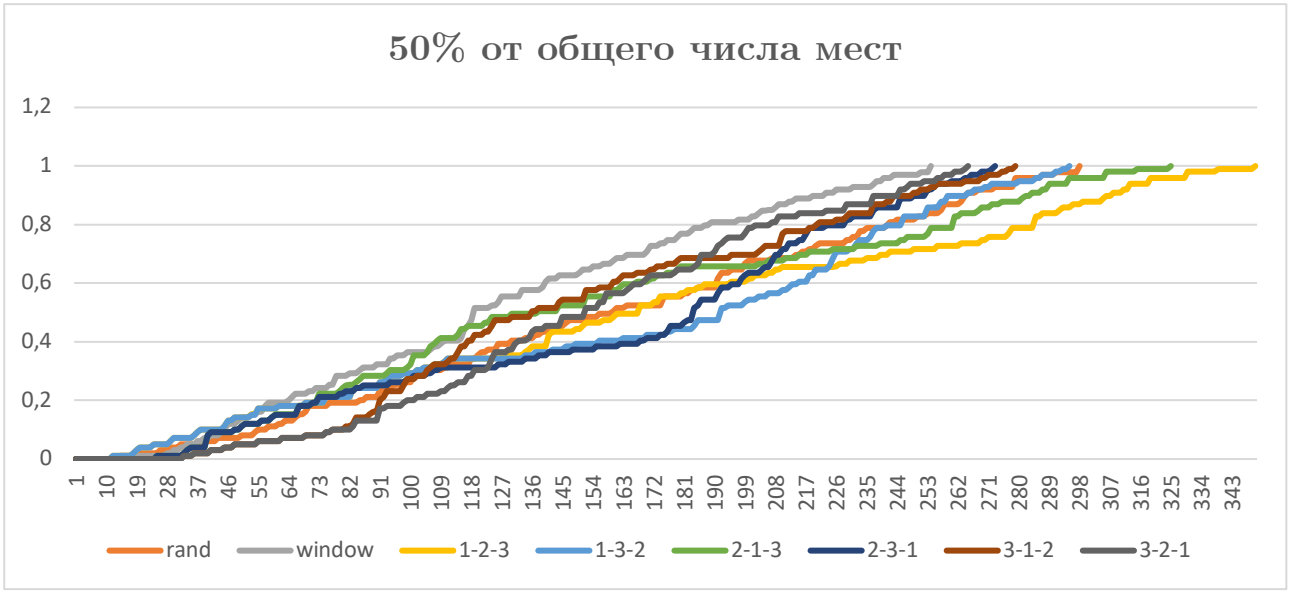


Диаграмма 3

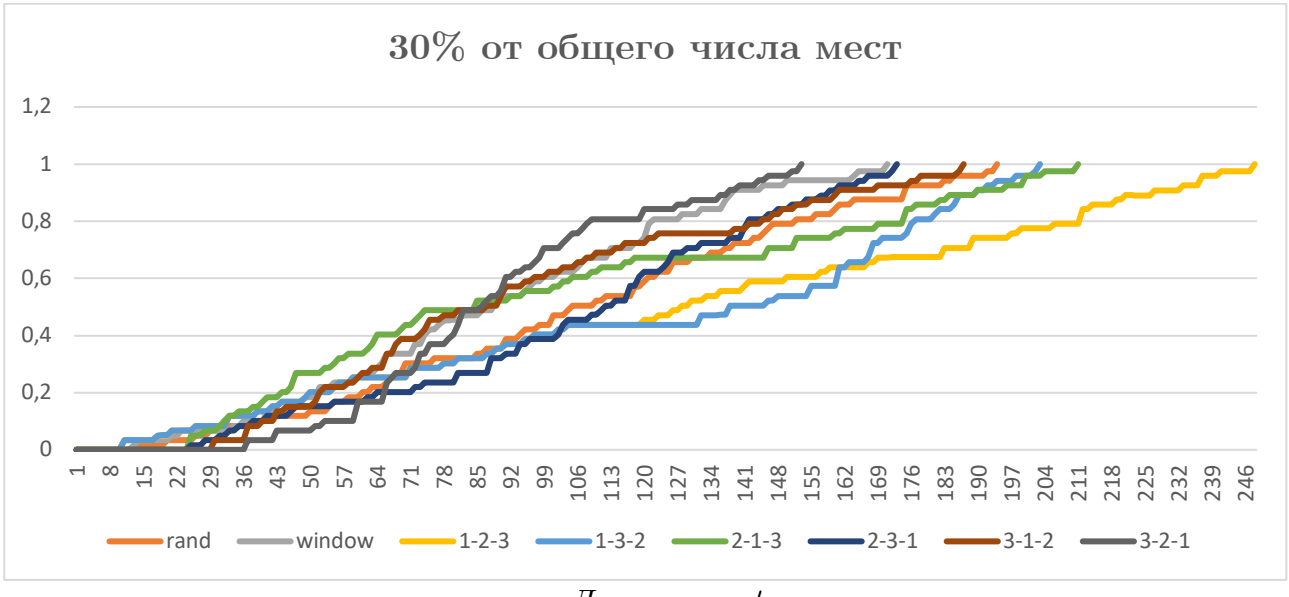


Диаграмма 4

*Время укладывания вещей на багажную полку на *диаграммах 2–4* выбиралось произвольно из диапазона от 2 до 4 секунд. Такое решение было принято, чтобы ускорить вычисления.
Из *диаграмм 2–4* можно выделить такую закономерность, что на малом числе пассажиров лучший результат показывает способ 3-2-1. Это можно обосновать тем, что при нем большее число людей может одновременно начать садиться.
Стоит также отметить, что по результатам данного эксперимента мы не можем точно сказать, какое время займет посадка, а можем лишь утверждать, что в некоторых ситуациях какой-то метод рассадки показывает себя лучше остальных.

4.3 Социальный эксперимент

При анализе процента пассажиров, не соблюдающих предписанный метод посадки, нам стало интересно, какой процент людей не следует правилам в реальной жизни. Для этого мы провели небольшое исследование в рамках школы, по результатам которого определили процент учеников, которые не соблюдают очередь при посещении столовой, их впоследствии будем называть нарушителями.

Ход исследования

1. С согласия администрации мы установили камеру в коридоре возле столовой на три дня (23.03–25.03). Мы выбрали эти дни, чтобы потом было время для анализа полученных данных.
2. 25.03 мы получили записи камер с трех обедов и приступили к их обработке.
3. Мы считали нарушителей (то есть тех, кто после мытья рук возвращался не в конец очереди, а в середину или начало), а также для получения процентного соотношения общее количество людей.
4. По итогу нами было отсмотрено около 45 минут (общий процесс обеда дольше, но очередь образуется только через несколько минут).
5. Нами получено, что нарушители составляют около 40% (43 человека из 101) от общего числа посетителей.

В рассматриваемом исследовании люди внутри очереди знакомы, поэтому количество нарушителей на наш взгляд больше, чем в очереди незнакомых людей, поэтому будем считать, что в очереди незнакомых людей всего $w\%$ нарушителей.

Кого считать нарушителем

Будем считать нарушителями тех, кто зашел в самолет не со своей группой. Это могут быть или люди, которые зашли вместе, хотя являются представителями разных групп (например, родитель с ребенком, сидящие рядом, хотя метод подразумевает то, что соседние места относятся к разным группам), или опоздавшие пассажиры, или пассажиры, случайным образом попавшие не в свою группу (возможно из-за ошибки при проверке билетов и делении на группы). При этом они все равно занимают именно свое место. Таким образом, понятно, что при неструктурированной посадке нарушителей нет (так как нет разбиения на группы)

Предлагаем теперь понаблюдать за тем, как меняется вид диаграммы при учете нарушителей (моделировать будем полную загрузку при времени на то, чтобы уложить багаж от 2 до 4 секунд)

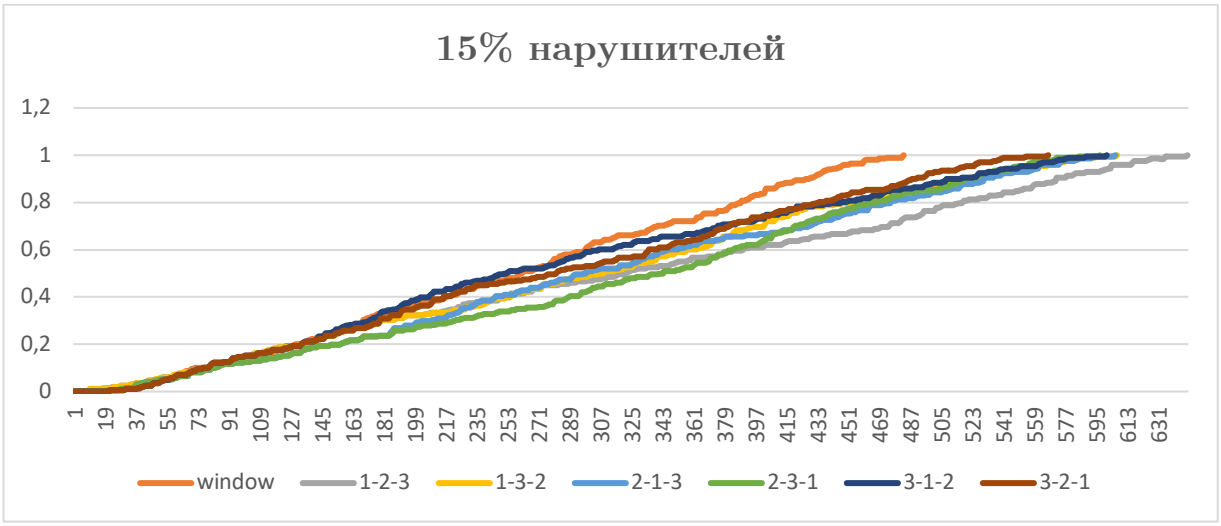


Диаграмма 5

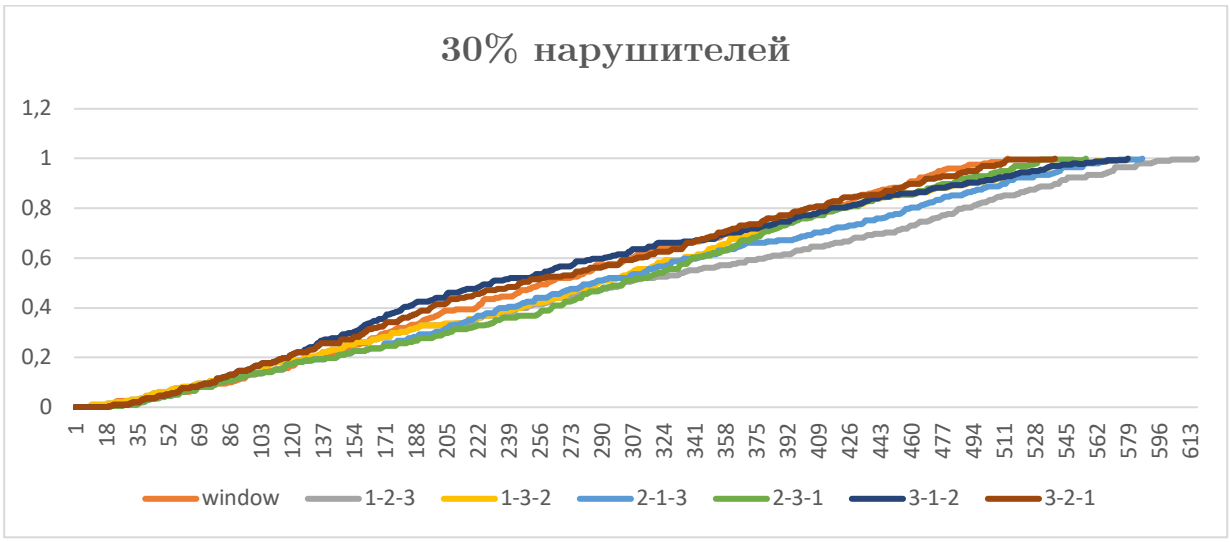


Диаграмма 6

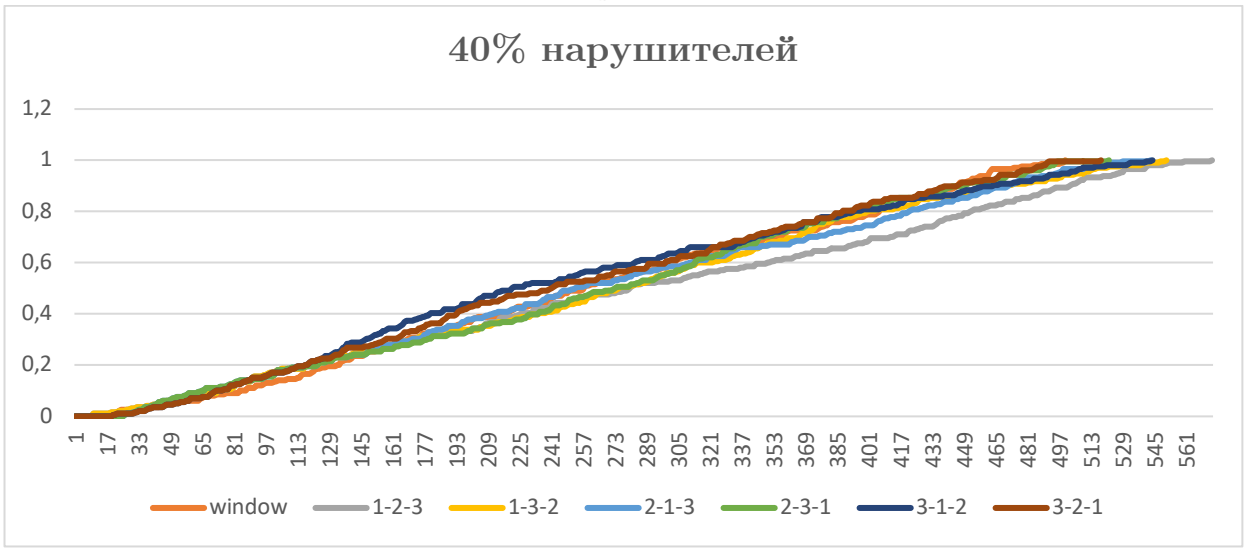


Диаграмма 7

Как мы видим из графиков, по мере увеличения числа нарушителей зависимости на диаграмме начинают стремиться к одной прямой - rand, при этом рассадка window по-прежнему, хоть на графике это незаметно, является самой быстрой. То, что данные методы рассадки при большом проценте нарушителей начинают стремиться к методу *rand* объясняется тем, что чем больше людей не соблюдают порядок, тем больше мест в самолете занимается в хаотичном порядке, не поддаваясь изначальному методу. А так как в реализации метод *rand* значительно проще (совсем не требует подготовки и отдельного разбиения на группы), то если очевидно, что будет большое количество нарушений (например, из-за экстренных событий или большого количества опаздывающих, когда люди начинают паниковать и торопиться), то стоит организовать посадку методом *rand* (легче, но при этом имеет ту же эффективность в рассматриваемом случае)

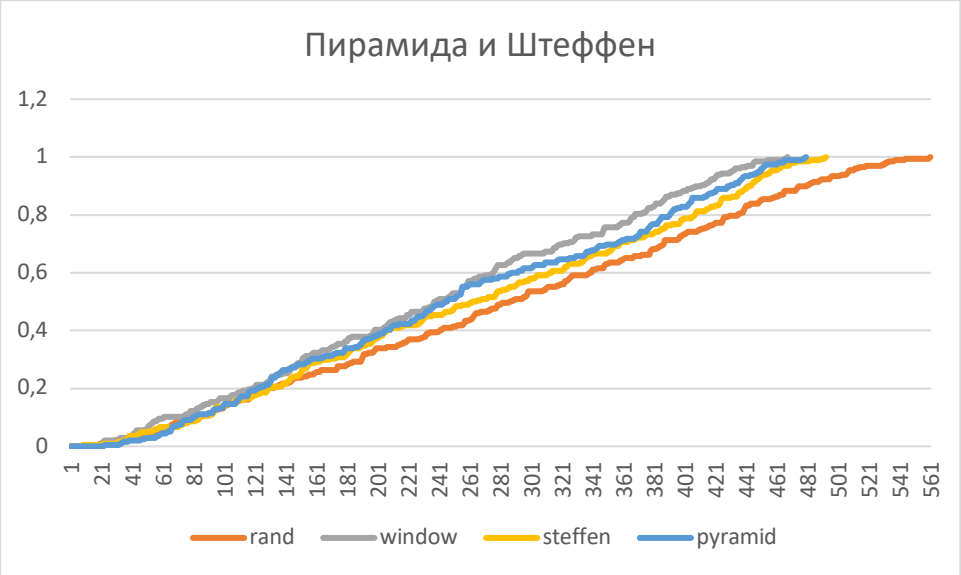
4.4 Дополнительные методы рассадки

Также нами было рассмотрено еще два метода посадки. Первый из них является упрощенным методом Штеффана, при котором сначала садятся нечетные ряды одной колонны, потом нечетные ряды другой, потом четные ряды опять первой колонны и, соответственно, в конце четные ряды второй (см. рис.1). Такой метод выигрышно проявляет себя, если в потоке людей часто встречаются группы знакомых, которые сразу занимают последовательные места, но если среди пассажиров знакомых мало, то такой метод рассадки будет показывать меньшие успехи

Рисунок 1

Рисунок 2

Второй метод является своеобразной пирамидкой. В этом методе салон самолета делится на 5 подгрупп, которые заполняются по порядку (см. рис.2). Такой метод также подвержен “проблемам друзей”, так как аналогично методу окно-середина-проход соседние по ряду места относятся к разным группам



4.5 Обобщающая модель

Теперь рассмотрим программу сеточного типа для посадки людей. В программе такого типа, считается, что любой объект (пассажир, проход, место) состоит из нескольких ячеек, что позволяет учитывать габариты людей и скорость их движения (за один ход сдвигаются на разное количество ячеек). Написанная нами программа является универсальной, то есть может быть использована при анализе самолетов с любым количеством проходов. Это достигается благодаря тому, что при написании программы считается, что самолет имеет один главный горизонтальный проход *main_road*, в который попадают при входе в самолет, а затем из *main_road* выходит столько вертикальных проходов, сколько рассматривается в данной модели, например, в самолете типа “Flying Wing” будет 4 вертикальных прохода, в самолете с тремя рядами сидений, описанном в условии, соответственно, два таких прохода, у узкофюзеляжного всего один.

Реализация этой модели такова:

1. У каждого из проходов есть координата начала (ось вдоль *main_road*). Также есть двумерный массив, состоящий из одномерных массивов, каждый из которых, хранит информацию о пассажирах, находящихся в одном из проходов (для каждого прохода свой массив). . Про каждого конкретного пассажира хранится информация о его состоянии (одно из 5 возможных: 0 - идёт по *main_road*, 1 - поворачивает, 2 - идёт к месту, 3 - складывает ручную кладь и садиться, 4 - сидит) и координата (ось вдоль прохода), причем количество людей в проходе может меняться, так как пассажиры могут входить в проход из *main_road* или уходить из него, занимая свое

место. Таким образом, с помощью рассмотренных координат на каждом шаге можно обращаться ко всем объектам (пассажирам) и менять их местоположение в соответствии с методом посадки.

2. Считается, что идти по проходу в ширину может только один человек (то есть идущего обгонять нельзя), но при этом если перед идущим человеком пассажир остановился и начал складывать багаж (то есть перешел в состояние 3), то первый может обогнать второго, причем делает он это с вдвое меньшей скоростью (в связи с необходимостью идти боком).

3. В случае блокировки мест метод работы полностью аналогичен рассмотренному в клеточном методе.

Эта программа очень удобна и позволяет анализировать самолеты почти всех типов. Если точнее, то всех тех, что могут уложиться в представленную схему:

Также нами был написан еще один код сеточного типа, который описывает высадку из самолета. Он схож по структуре с программой, описывающей посадку, но при этом имеет некоторые отличия:

1. Отдельно проверяется возможность выхода в главный проход (далее в статье, как и в самом коде, *buffer*)
2. Люди не могут выходить с заблокированных мест, то есть если занято место у входа или по центру, то пассажир, сидящий у окна, выйти не может.

3. Также после каждого шага проверяется, может ли кто-то из сидящих вклиниться в вертикальный проход (то есть проверяется расстояние между двумя уже идущими людьми). Если да, то он встает в проход до начала следующего шага движения.

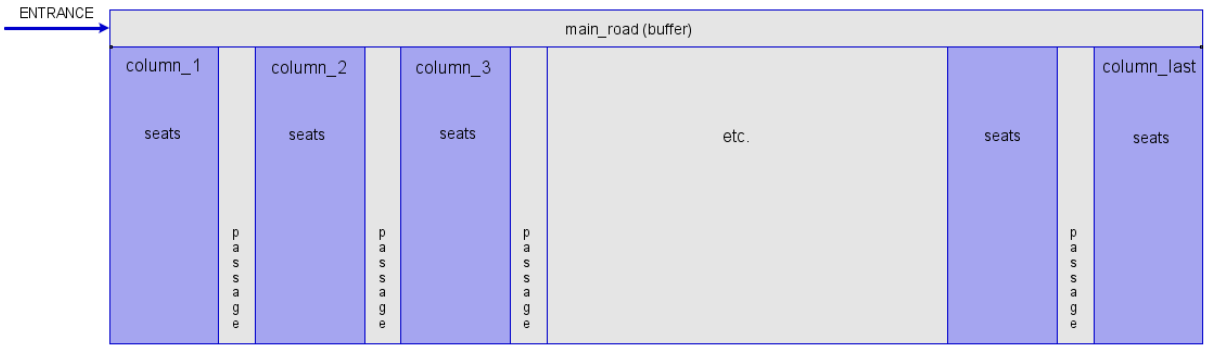


Схема самолета, рассматриваемого при написании программы сеточного типа

4.6 Тестирование

Данный вариант модели писался позже *simple_ model,* поэтому ряд исследований мы решили опустить и оставили только главное: оценим, сколько времени займет посадка случайным методом. Просмотрев записи реальных посадок, мы оценили

примерное время, которое человек в среднем тратит на то, чтобы сесть и положить багаж, оно оказалось равным приблизительно 3 и 8 секунд соответственно, проинициализируем параметры этими данными и найдем время, требующееся на полную загрузку воздушного судна. Здесь мы приведем только готовый результат. Итак, для модели с двумя проходами получаем время 2289 секунд, для узкофюзеляжного варианта - 1579 секунд, и для "Flying Wing" - 1493

5 Обсуждение

Наша команда не учла некоторых факторов, которые имеют место быть в реальном процессе. К примеру, люди могут путешествовать семьями или группами, которые можно рассматривать как большие неделимые объекты в программе, причем время посадки в таком случае может как уменьшиться, так и увеличиться. Также при высадке был принят ряд допущений, которые смотрятся вполне логично, но при этом не всегда выполняются. Так, мы полагали, что вставать в проход объект может только тогда, когда все кресла от него и до прохода свободны, но, вообще говоря, это необязательно, достаточно привести пример поля с большими расстояниями между соседними рядами кресел. Помимо этого, зачастую при посадке в проходах могут находиться члены экипажа, которые помогают пассажирам разместить их багаж. Однако мы приняли решение не моделировать настолько тонкие и непростые в реализации идеи, а сосредоточиться на построении работающей модели, результат работы которой можно увидеть.

Так как большая часть задач сводилась к поиску оптимального решения, а не к точному определению времени посадки, можно утверждать, что разработанная модель с хорошей точностью может дать нужные ответы.

Стоит также сказать пару слов о высадке пассажиров. Мы считаем, что процесс посадки протекает идентично процессу высадки с точностью до обратной перемотки, поэтому найденный оптимальный вариант посадки применим и для высадки

6 Список ресурсов

1. Обсуждение принципов посадок на борт самолета: https://travelask.ru/questions/13233-po-kakomu-printsipu-proishodit-posadka-na-bort-samolyota (дата обращения 22.03)
2. The Better Boarding Method Airlines Won't Use: https://www.youtube.com/watch?v=oAHbLRjF0vo (дата обращения 22.03)
3. Sveinung Erland, Jevgenijs Kaupužs, Vidar Frette, Rami Pugatch, Eitan Bachmat "Lorentzian-geometry-based analysis of airplane boarding policies highlights “slow passengers first” as better" (дата обращения 23.03)
4. Схема салона Boeing 737-800 Ryanair: https://mirputeshestvij.mediasole.ru/shema_salona_boeing_737800__ryanair_luchshie_mesta_v_samolet (дата обращения 23.03).
5. Анимации посадок: https://drive.google.com/drive/folders/14mPvXcbpG6Q_qly7M0X8nXgxOmlTrS4D?usp=sharing

7 Приложения

Простая модель

```
import random


class passenger():
    def __init__(self, number):
        global ambition_list
        global plane
        random.seed(1)
        self.number = number
        self.ambition = ambition_list[self.number]
        self.position = [0, 3]
        plane[self.position[0]][self.position[1]] = 1
        self.other_time_for_feet = random.choice([1, 2, 3]) # от 1 до 3
        self.other_time_for_bags = random.choice([2, 3, 4]) # от 2 до 4
        self.time_to_stand = 0
        self.speed = 1
        self.condition = 0 # 0 - идёт,  1 - садится, 2 - сидит

    def move(self):
        global plane
        global other_time_list
        global conditions_list
        if (self.condition == 0) and (self.position[0] == self.ambition[0]):
            self.condition = 1
            conditions_list[self.position[0]][self.position[1]] =
self.condition
            self.time_to_stand = self.other_time_for_feet +
self.other_time_for_bags
            additional_time = 0
            if self.ambition[1] > 3:
                for i in range(4, self.ambition[1], 1):
                    if (other_time_list[self.position[0]][i] != 0):
                        additional_time +=
other_time_list[self.position[0]][i]
            if self.ambition[1] < 3:
                for i in range(2, self.ambition[1], -1):
                    if (other_time_list[self.position[0]][i] != 0):
                        additional_time +=
other_time_list[self.position[0]][i]
            self.time_to_stand += additional_time * 2
        elif self.condition == 0:
            if (plane[self.position[0] + 1][self.position[1]] == 0):
                plane[self.position[0]][self.position[1]] = 0
                conditions_list[self.position[0]][self.position[1]] = 3
                self.position[0] += 1
                plane[self.position[0]][self.position[1]] = 1
                conditions_list[self.position[0]][self.position[1]] =
self.condition
        elif self.condition == 1:
            self.time_to_stand -= 1
            if self.time_to_stand <= 0:
                plane[self.position[0]][self.position[1]] = 0
                conditions_list[self.position[0]][self.position[1]] = 3
                self.position = self.ambition
                plane[self.position[0]][self.position[1]] = 1
```

```
                self.condition = 2
                conditions_list[self.position[0]][self.position[1]] =
self.condition
                other_time_list[self.position[0]][self.position[1]] =
self.other_time_for_feet

    def get_condition(self):
        return self.condition


def let_one_in():
    global on_board_now
    global passenger_list
    global plane
    if plane[0][3] == 0 and on_board_now < len(ambition_list):
        passenger_list.append(passenger(on_board_now))
        on_board_now += 1


def sort_random(n):
    global ambition_list
    global number_of_seats
    ambition_list = []
    for i in range(1, number_of_seats + 1):
        for j in range(0, 7):
            if j == 3:
                continue
            ambition_list.append([i, j])
    random.shuffle(ambition_list)
    ambition_list = ambition_list[0: int(number_of_seats * 6 * n)]


def sort_sections(n, p1, p2, p3):
    global ambition_list
    global number_of_seats
    ambition_list = []
    for i in range(1, number_of_seats + 1):
        for j in range(0, 7):
            if j == 3:
                continue
            ambition_list.append([i, j])
    random.shuffle(ambition_list)
    ambition_list = ambition_list[0: int(number_of_seats * 6 * n)]

    parts = [[], [], []]
    for i in range(0, len(ambition_list)):
        if ambition_list[i][0] <= 11:
            parts[0].append(ambition_list[i])
        elif ambition_list[i][0] <= 22:
            parts[1].append(ambition_list[i])
        elif ambition_list[i][0] <= 33:
            parts[2].append(ambition_list[i])
    random.shuffle(parts[0])
    random.shuffle(parts[1])
    random.shuffle(parts[2])
    ambition_list = (parts[p1 - 1] + parts[p2 - 1] + parts[p3 - 1])


def sort_windows(n):
```

```
    global ambition_list
    global number_of_seats
    ambition_list = []
    for i in range(1, number_of_seats + 1):
        for j in range(0, 7):
            if j == 3:
                continue
            ambition_list.append([i, j])
    random.shuffle(ambition_list)
    ambition_list = ambition_list[0: int(number_of_seats * 6 * n)]
    parts = [[], [], []]
    for i in range(0, len(ambition_list)):
        if ambition_list[i][1] == 0 or ambition_list[i][1] == 6:
            parts[0].append(ambition_list[i])

        if ambition_list[i][1] == 1 or ambition_list[i][1] == 5:
            parts[1].append(ambition_list[i])

        if ambition_list[i][1] == 2 or ambition_list[i][1] == 4:
            parts[2].append(ambition_list[i])
    random.shuffle(parts[0])
    random.shuffle(parts[1])
    random.shuffle(parts[2])
    ambition_list = (parts[0] + parts[1] + parts[2])


def sort_steffen(n):
    global ambition_list
    global number_of_seats
    ambition_list = []
    g_1 = []
    g_2 = []
    g_3 = []
    g_4 = []
    for i in range(1, number_of_seats + 1, 2):
        g_1.append([i, 0])
        g_1.append([i, 1])
        g_1.append([i, 2])

        g_2.append([i, 4])
        g_2.append([i, 5])
        g_2.append([i, 6])
    for i in range(2, number_of_seats + 1, 2):
        g_3.append([i, 0])
        g_3.append([i, 1])
        g_3.append([i, 2])

        g_4.append([i, 4])
        g_4.append([i, 5])
        g_4.append([i, 6])
    random.shuffle(g_1)
    random.shuffle(g_2)
    random.shuffle(g_3)
    random.shuffle(g_4)
    ambition_list = g_1 + g_2 + g_3 + g_4


def sort_piramidka(n):
    global ambition_list
```

```
    global number_of_seats
    ambition_list = []
    g_1 = []
    g_2 = []
    g_3 = []
    g_4 = []
    g_5 = []
    for i in range(1, number_of_seats + 1):
        for j in range(0, 7):
            if i >= 13 and (j == 0 or j == 6):
                g_1.append([i, j])
            elif (i >= 6 and i <= 12 and (j == 0 or j == 6)) or (i >= 13 and
i <= 33 and (j == 1 or j == 5)):
                g_2.append([i, j])
            elif (i >= 1 and i <= 5 and (j == 0 or j == 6)) or (i >= 6 and i
<= 22 and (j == 1 or j == 5)):
                g_3.append([i, j])
            elif (i >= 1 and i <= 5 and (j == 1 or j == 5)) or (i >= 22 and i
<= 33 and (j == 2 or j == 4)):
                g_4.append([i, j])
            elif i <= 21 and (j == 2 or j == 4):
                g_5.append([i, j])
    random.shuffle(g_1)
    random.shuffle(g_2)
    random.shuffle(g_3)
    random.shuffle(g_4)
    random.shuffle(g_5)
    ambition_list = g_1 + g_2 + g_3 + g_4 + g_5


def sort_steffen_and_group(n):
    global ambition_list
    global number_of_seats
    ambition_list = []
    g_1 = []
    g_2 = []
    g_3 = []
    g_4 = []
    for i in range(1, number_of_seats + 1, 2):
        g_1.append([i, 0])
        g_1.append([i, 1])
        g_1.append([i, 2])

        g_2.append([i, 4])
        g_2.append([i, 5])
        g_2.append([i, 6])
    for i in range(2, number_of_seats + 1, 2):
        g_3.append([i, 0])
        g_3.append([i, 1])
        g_3.append([i, 2])

        g_4.append([i, 4])
        g_4.append([i, 5])
        g_4.append([i, 6])
    random.shuffle(g_1)
    random.shuffle(g_2)
    random.shuffle(g_3)
    random.shuffle(g_4)
    ambition_list = g_1 + g_2 + g_3 + g_4
```

```
for iteration in range(0, 1000):
    number_of_seats = 33
    plane = [] # список с координатами
    for i in range(0, number_of_seats + 1):
        plane.append([0, 0, 0, 0, 0, 0, 0]) # проход с индексом 0
    other_time_list = []
    for i in range(0, number_of_seats + 1):
        other_time_list.append([0, 0, 0, 0, 0, 0, 0])
    conditions_list = []
    for i in range(0, number_of_seats + 1):
        conditions_list.append([3, 3, 3, 3, 3, 3, 3])
    ambition_list = []
    ################################
    # Выбор метода сортировки condition_list
    # sort_random(0.3)
    # sort_sections(1, 3, 2, 1)
    # sort_windows(0.3)
    # sort_piramidka(0.3)
    # sort_steffen(0.3)
    ################################
    on_board_now = 0
    passenger_list = []
    passenger_list.append(passenger(on_board_now))
    on_board_now += 1
    sat_down_now = 0
    sat_down_step_forward = 0
    step_now = 1
    with open("data/simple_model_out/out_of_iteration_" + str(iteration) +
".csv", "w") as f:
        while (sat_down_now != len(ambition_list)):
            # Файловый вывод
            #f.write(str(step_now) + ";" + str(sat_down_now) + "\n")
            let_one_in()
            sat_down_step_forward = 0
            for i in range(0, len(passenger_list)):
                passenger_list[i].move()
                if (passenger_list[i].get_condition() == 2):
                    sat_down_step_forward += 1
            if (sat_down_step_forward != sat_down_now):
                #print(sat_down_step_forward)
                pass
            step_now += 1
            sat_down_now = sat_down_step_forward
            # f.write(str(step_now) + ";" + str(sat_down_now) + "\n")
            # Вывод результата в файл
            # s = ""
            # for i in range(0, 34):
            #     for j in range(0, 7):
            #         s += str(conditions_list[i][j])
            # f.write(s + "\n")
```

Обобщенная модель

```
import random

ambition_list = []

list_of_passengers = []
list_of_roads = []
list_of_seats = []

main_road = []
ambition_list = []

list_of_moving_points = []
list_of_seats_coordinates = []

road_wight = 5
seats_width = 6
seats_lenght = 6
distance_between_seats = 3


class passenger():
    def __init__(self, number):
        global ambition_list
        global main_road
        self.number = number
        self.position = -1
        self.visible_position = [self.number, 0, 0]
        self.long = 3
        self.speed = 3
        self.time_to_sit = 3
        self.time_to_bags = 8

        self.oll_time = self.time_to_sit + self.time_to_bags # общее время
сесть
        self.condition = 0 # 0 - идёт по главной, 1 - поворачивает, 2 идёт к
месту, 3 - садится, 4 - сидит

        self.ambition = ambition_list[self.number]

        main_road.append(self.visible_position)

    def move(self):
        global list_of_roads
        global main_road

        global sat_down

        global list_of_seats

        global list_of_moving_points
        global list_of_seats_coordinates
        #print(list_of_roads)
        #print(self.visible_position)
        if self.condition == 0:
            # идет по главной
            # определение максимальной длины шага
            max_step_long = 0
```

```
            for i in range(0, len(main_road)):
                if main_road[i][0] == self.number:
                    break
            if len(main_road) > i + 1:
                if list_of_passengers[main_road[i + 1][0]].get_last_point() >
self.visible_position[1] + self.speed:
                    max_step_long = self.speed
                else:
                    max_step_long = (list_of_passengers[main_road[i +
1][0]].get_last_point() - self.visible_position[1])
            else:
                max_step_long = self.speed

            # проверка на выход
            if list_of_moving_points[self.ambition[0]] - max_step_long <=
self.visible_position[1]:
                # шаг ровно к точке входа
                self.condition = 1
                self.visible_position[2] = self.condition
                self.visible_position[1] =
list_of_moving_points[self.ambition[0]]
            else:
                # обычный шаг
                self.visible_position[1] += max_step_long

        elif (self.condition == 1):
            # готов повернуть
            # считаем, что проход недостаточно широкий, чтобы идти в 2
потока, но достаточно широкий, чтобы не замедлять
            if len(list_of_roads[self.ambition[0]]) != 0:
                #print(list_of_roads[self.ambition[0]][0])
                if
list_of_passengers[list_of_roads[self.ambition[0]][0][0]].get_last_point() >=
road_wight:
                    '''переносим'''
                    self.condition = 2
                    self.position = self.ambition[0]
                    self.visible_position[1] = 0
                    self.visible_position[2] = self.condition
                    for i in range(0, len(main_road)):
                        if main_road[i][0] == self.number:
                            main_road.pop(i)
                            break

list_of_roads[self.ambition[0]].append(self.visible_position)
                else:
                    # ждем
                    pass
            else:
                # переносим
                self.condition = 2
                self.position = self.ambition[0]
                self.visible_position[1] = 0
                self.visible_position[2] = self.condition
                for i in range(0, len(main_road)):
                    if main_road[i][0] == self.number:
                        main_road.pop(i)
                        break
                list_of_roads[self.ambition[0]].append(self.visible_position)
```

```
        elif self.condition == 2:
            # идет к месту
            max_step_long = 0
            for i in range(0, len(list_of_roads[self.position])):
                if list_of_roads[self.position][i][0] == self.number:
                    break
            if len(list_of_roads[self.position]) > i + 1:
                if list_of_passengers[list_of_roads[self.position][i +
1][0]].get_last_point() > self.visible_position[1] + self.speed:
                    max_step_long = self.speed
                else:
                    if list_of_passengers[list_of_roads[self.position][i +
1][0]].get_condition() == 3:
                        if len(list_of_roads[self.position]) > i + 2:
                            if
list_of_passengers[list_of_roads[self.position][i + 2][0]].get_last_point() >
self.visible_position[1] + self.speed // 2:
                                max_step_long = self.speed // 2
                            else:
                                max_step_long =
(list_of_passengers[list_of_roads[self.position][i + 2][0]].get_last_point()
- self.visible_position[1])
                        else:
                            max_step_long = self.speed // 2
                    else:
                        max_step_long =
(list_of_passengers[list_of_roads[self.position][i + 1][0]].get_last_point()
- self.visible_position[1])


            else:
                max_step_long = self.speed

            # проверка на выход

            if (self.ambition[1] - max_step_long <=
self.visible_position[1]):
                # шаг ровно к точке входа
                self.condition = 3
                self.visible_position[2] = self.condition
                self.visible_position[1] = self.ambition[1]

                for i in range(0, len(list_of_seats)):
                    if (list_of_seats[i][0] == self.ambition[0] and
list_of_seats[i][1] == self.ambition[1]):
                        if (self.ambition[2] > 3 and list_of_seats[i][2] > 3
and list_of_seats[i][0] < self.ambition[2]):
                            self.oll_time += list_of_seats[i][3]
                        elif (self.ambition[2] < 3 and list_of_seats[i][2] <
3 and list_of_seats[i][0] > self.ambition[2]):
                            self.oll_time += list_of_seats[i][3]
            else:
                # обычный шаг
                self.visible_position[1] += max_step_long

        elif (self.condition == 3):
            self.oll_time -= 1
```

```
            if (self.oll_time <= 0):
                self.condition = 4

                list_of_seats.append([self.ambition[0], self.ambition[1],
self.ambition[2], self.time_to_sit])

                sat_down += 1
                self.position = self.ambition[0]
                self.visible_position[2] = self.condition
                for i in range(0, len(list_of_roads[self.ambition[0]])):
                    if (list_of_roads[self.ambition[0]][i][0] ==
self.number):
                        list_of_roads[self.ambition[0]].pop(i)
                        break

        else:
            pass

    def get_last_point(self):
        return self.visible_position[1] - self.long

    def get_condition(self):
        return self.condition


def let_one_in():
    # Пустить ещё кого-нибудь
    global on_board_now

    global list_of_passengers
    global ambition_list

    global main_road

    if len(ambition_list) > on_board_now:
        if len(main_road) != 0:
            if list_of_passengers[main_road[0][0]].get_last_point() >=
road_wight:
                list_of_passengers.append(passenger(on_board_now))
                on_board_now += 1
        else:
            list_of_passengers.append(passenger(on_board_now))
            on_board_now += 1


def take_a_step():

    for i in range(0, len(list_of_roads)):
        list_of_roads[i].sort(key=lambda x: x[0])
        list_of_roads.reverse()

    main_road.sort(key=lambda x: x[0])
    main_road.reverse()
    print(list_of_roads[0])

    for x in range(0, len(list_of_roads)):
        i = 0
        n = len(list_of_roads[x])
        while i < n:
```

```
            if len(list_of_roads[x]) != 0:
                list_of_passengers[list_of_roads[x][i][0]].move()
                if n == len(list_of_roads[x]):
                    i += 1
                else:
                    n = len(list_of_roads[x])

    i = 0
    n = len(main_road)
    while i < n:
        list_of_passengers[main_road[i][0]].move()
        if n == len(main_road):
            i += 1
        else:
            n = len(main_road)


def build_for_flying_wing():
    # Собрать параметры для летающего крыла
    global ambition_list

    global list_of_roads
    global main_road
    global list_of_seats

    global list_of_moving_points
    global list_of_seats_coordinates

    global road_wight
    global seats_width
    global seats_lenght
    global distance_between_seats


    list_of_roads.append([])
    list_of_roads.append([])
    list_of_roads.append([])
    list_of_roads.append([])

    list_of_moving_points.append(seats_width * 3)
    list_of_moving_points.append(seats_width * 3 * 3 + road_wight)
    list_of_moving_points.append(seats_width * 3 * 5 + road_wight * 2)
    list_of_moving_points.append(seats_width * 3 * 7 + road_wight * 3)


def build_for_narrow_body():
    # Собрать параметры для узкофюзеляжного
    global ambition_list

    global list_of_roads
    global main_road
    global list_of_seats

    global list_of_moving_points
    global list_of_seats_coordinates

    global road_wight
    global seats_width
    global seats_lenght
```

```
    global distance_between_seats

    list_of_roads.append([])

    list_of_moving_points.append(seats_width * 3)


def build_for_two_entrance():
    # Собрать параметры для широкого с двумя рядами
    global ambition_list

    global list_of_roads
    global main_road
    global list_of_seats

    global list_of_moving_points
    global list_of_seats_coordinates

    global road_wight
    global seats_width
    global seats_lenght
    global distance_between_seats

    list_of_roads.append([])
    list_of_roads.append([])

    list_of_moving_points.append(seats_width * 3)
    list_of_moving_points.append(seats_width * 3 * 3 + road_wight)


def build_ambition_list_for_flying_wing_random(n):
    # Массив с билетами для летающего крыла (случайно)
    global ambition_list
    global distance_between_seats
    global seats_lenght

    for i in range(0, 4):
        for j in range(0, 14):
            for k in range(0, 7):
                if (k == 3):
                    continue
                if (i == 0 and j >= 11 and k < 3):
                    continue
                if (i == 3 and j >= 11 and k > 3):
                    continue
                ambition_list.append([i, j * (seats_lenght +
distance_between_seats), k])

    random.shuffle(ambition_list)
    ambition_list = ambition_list[0: int(len(ambition_list) * n)]


def build_ambition_list_for_narrow_body_random(n):
    # Массив с билетами для узкофюзеляжного (случайно)
    global ambition_list
    global distance_between_seats
    global seats_lenght

    for i in range(0, 33):
```

```
        for j in range(0, 7):
            if j == 3:
                continue
            ambition_list.append([0, i * (seats_lenght +
distance_between_seats), j])

    random.shuffle(ambition_list)
    ambition_list = ambition_list[0: int(len(ambition_list) * n)]


def build_ambition_list_for_two_entrance_random(n):
    # Массив с билетами для широкого с двумя рядами (случайно)
    global ambition_list
    global distance_between_seats
    global seats_lenght

    for i in range(0, 2):
        for j in range(0, 20):
            for k in range(0, 7):
                if k == 3:
                    continue
                if i == 0 and j >= 11 and k < 3:
                    continue
                if i == 3 and j >= 11 and k > 3:
                    continue
                ambition_list.append([i, j * (seats_lenght +
distance_between_seats), k])

    random.shuffle(ambition_list)
    ambition_list = ambition_list[0: int(len(ambition_list) * n)]


def build_ambition_list_for_narrow_body_sections(n, p1, p2, p3):
    # Массив с билетами для узкофюзеляжного (по группам)
    global ambition_list
    global distance_between_seats
    global seats_lenght


for iteration in range(0, 1000):

    ambition_list = []

    list_of_passengers = []
    list_of_roads = []

    list_of_seats = []

    main_road = []
    ambition_list = []

    list_of_moving_points = []
    list_of_seats_coordinates = []




    ##################################################
    build_for_flying_wing()
```

```
    # build_for_narrow_body()
    # build_for_two_entrance()
    ##################################################

    ##################################################
    build_ambition_list_for_flying_wing_random(1)
    # build_ambition_list_for_narrow_body_random(1)
    # build_ambition_list_for_two_entrance_random(1)
    ##################################################



    on_board_now = 0
    sat_down = 0
    step = 0
    with open("main_model_out/out_of_iteration_" + str(iteration) + ".csv",
"w") as f:
        while sat_down < len(ambition_list):
            step += 1
            let_one_in()
            take_a_step()
            print(sat_down)

            f.write(str(step) + ";" + str(sat_down) + "\n")
```

#Визуализация посадки

```
from tkinter import *
from random import *
import time


class model(Frame):
    def __init__(self, q):
        super().__init__()
        self.get_graphics(q)


    def get_graphics(self, q):

        self.master.title("Plane")
        self.pack(fill=BOTH, expand=1)
        canvas = Canvas(self, width=1450, height=310, bg="gray80")

        x_help = 15
        y_help = 15
        for x in range(0, len(q)):
            for y in range(0, 7):
                if q[x][y] == "0":
                    canvas.create_rectangle(
                    x_help + x * 42, y_help + y * 42, x_help + x * 42 + 35,
y_help + y * 42 + 35,
                    outline="gray0", fill="salmon")

                elif q[x][y] == "1":
                    canvas.create_rectangle(
                    x_help + x * 42, y_help + y * 42, x_help + x * 42 + 35,
y_help + y * 42 + 35,
```

```
                    outline="gray0", fill="cyan4")

                elif q[x][y] == "2":
                    canvas.create_rectangle(
                    x_help + x * 42, y_help + y * 42, x_help + x * 42 + 35,
y_help + y * 42 + 35,
                    outline="gray0", fill="grey52")

                else:
                    if y == 3:
                        canvas.create_rectangle(
                        x_help + x * 42, y_help + y * 42, x_help + x * 42 +
35, y_help + y * 42 + 35,
                        outline="gray80", fill="gray80")

                    else:
                        canvas.create_rectangle(
                        x_help + x * 42, y_help + y * 42, x_help + x * 42 +
35, y_help + y * 42 + 35,
                        outline="gray0", fill="azure")

        canvas.pack(fill=BOTH, expand=1)


root = Tk()
root.geometry("1450x350" + "+" + str(10) + "+" + str(50))
root.title("Plane")
root.configure(bg='Snow')
root.resizable(width=False, height=False)

f = open("data/simple_model_out/out_of_iteration_0.csv", "r").readlines()
for i in range(0, len(f)):
    q = []
    for j in range(0, 34):
        w = []
        for k in range(0, 7):
            w.append(f[i][7 * j + k])
        q.append(w)
    print(q)
    win_1 = model(q).place(x = 20, y = 10)
    root.update()
    time.sleep(0.005)

root.mainloop()
```

Высадка

```
# model of disembarking in a general case
import random

WIDTH_AISLE = 5
passengers_on_board = 32
aisles = []
buffer = []


class Aisle:
    def __init__(self, seating, y):
        self.seating = seating
        self.y = y
        self.standing = []

    # insert passenger in an aisle
    def insertion_in_aisle(self):
        for i in range(len(self.seating)):
            # aisle end coordinates
            x1 = WIDTH_AISLE * i
            x2 = WIDTH_AISLE * (i + 1)
            is_in_range = False
            for elem in self.standing:
                if x1 <= elem.x < x2 or x1 < elem.x + WIDTH_AISLE < x2:
                    is_in_range = True
            if not is_in_range:
                inx_aisle = self.seating[i].index([])
                inx_desired = -1
                inx_left = -1
                inx_right = -1
                for j in range(inx_aisle - 1, -1, -1):
                    if self.seating[i][j] != 0:
                        inx_left = j
                        break
                for j in range(inx_aisle + 1, len(self.seating[i])):
                    if self.seating[i][j] != 0:
                        inx_right = j
                        break
                if inx_right == -1 and inx_left != -1:
                    inx_desired = inx_left
                elif inx_right != -1 and inx_left == -1:
                    inx_desired = inx_right
                elif inx_right != -1 and inx_left != -1:
                    if self.seating[i][inx_left].aggression >
self.seating[i][inx_right].aggression:
                        inx_desired = inx_left
                    else:
                        inx_desired = inx_right
                inx_insert_in_standing = -1
                for j in range(len(self.standing)):
                    if self.standing[j].x < x1:
                        inx_insert_in_standing = j + 1
                if inx_desired != -1:
                    self.seating[i][inx_desired].state = 1
                    self.seating[i][inx_desired].x = x1
                    self.seating[i][inx_desired].y = self.y * WIDTH_AISLE
```

```
                    self.standing.insert(inx_insert_in_standing,
self.seating[i][inx_desired])
                    self.seating[i].pop(inx_desired)
                    self.seating[i].insert(inx_desired, 0)

    # do something on the passenger depending on his state
    def move(self):
        global buffer
        index = {'to_insert': -1, 'to_remove': 0}
        for i in range(len(self.standing)):
            if self.standing[i].state == 1 and self.standing[i].time_to_stand
> 0:
                self.standing[i].time_to_stand -= 1
            elif self.standing[i].state == 1 and
self.standing[i].time_to_stand == 0:
                if self.standing[i].time_to_get_lug == 0:
                    self.standing[i].state = 3
                else:
                    self.standing[i].state = 2
                    self.standing[i].time_to_get_lug -= 1

            if self.standing[i].state == 2 and
self.standing[i].time_to_get_lug > 0:
                self.standing[i].time_to_get_lug -= 1
            elif self.standing[i].state == 2 and
self.standing[i].time_to_get_lug == 0:
                if i == len(self.standing) - 1:
                    self.standing[i].state = 3
                else:
                    if self.standing[i + 1].state != 4:
                        self.standing[i].state = 3

            if self.standing[i].state == 3:
                if i == 0:
                    if self.standing[i].x == 0:
                        is_busy = False
                        for elem in buffer:
                            if self.standing[i].y + WIDTH_AISLE > elem.y >
self.standing[i].y \
                                    or self.standing[i].y + WIDTH_AISLE >
elem.y + WIDTH_AISLE > self.standing[i].y:
                                is_busy = True
                        if not is_busy:
                            inx = 0
                            for j in range(len(buffer)):
                                if buffer[j].y < self.standing[i].y:
                                    inx = j + 1
                            index['to_insert'] = inx
                    elif self.standing[i].x - self.standing[i].speed > 0:
                        self.standing[i].x = self.standing[i].x -
self.standing[i].speed
                    elif self.standing[i].x - self.standing[i].speed <= 0:
                        self.standing[i].x = 0
                else:
                    if self.standing[i].x - self.standing[i].speed >=
self.standing[i - 1].x + WIDTH_AISLE:
                        self.standing[i].x = self.standing[i].x -
self.standing[i].speed
```

```
                    elif self.standing[i].x - self.standing[i].speed <
self.standing[i - 1].x + WIDTH_AISLE:
                        self.standing[i].x = self.standing[i - 1].x +
WIDTH_AISLE
                        # self.standing[i].state = 4 TODO

            if self.standing[i].state == 4:
                pass

        if index['to_insert'] != -1:
            self.standing[0].y = self.y
            buffer.insert(index['to_insert'], self.standing[0])
            self.standing.pop(0)
        self.insertion_in_aisle()


class Passenger:
    # state = 0 means passenger is seating, state = 1 --- is standing
    # state = 2 --- is getting luggage, state = 3 --- is going, 4 --- is
overtaking
    def __init__(self, speed, time_to_stand, time_to_get_lug, aggression,
x=0, y=0):
        self.speed = speed
        self.time_to_stand = time_to_stand
        self.time_to_get_lug = time_to_get_lug
        self.state = 0
        self.aggression = aggression
        self.x = x
        self.y = y


def move_in_buffer():
    global buffer
    global passengers_on_board
    to_delete = []
    for i in range(len(buffer)):
        if i == 0:
            if buffer[i].y < 0:
                to_delete.append(i)
                passengers_on_board -= 1
            else:
                buffer[i].y -= buffer[i].speed
        else:
            if i - 1 not in to_delete:
                if buffer[i].y - buffer[i].speed < buffer[i - 1].y +
WIDTH_AISLE:
                    buffer[i].y = buffer[i - 1].y + WIDTH_AISLE
                else:
                    buffer[i].y -= buffer[i].speed
            else:
                if buffer[i].y - buffer[i].speed < 0:
                    to_delete.append(i)
                else:
                    buffer[i].y -= buffer[i].speed
    for i in range(len(to_delete)):
        buffer.pop(to_delete[i] - i)


sec = 0
```

```
# Здесь задаются ряды
aisles.append(Aisle(seating=[[Passenger(1, 1, 1, 1), [], Passenger(1, 1, 1,
1), Passenger(1, 1, 1, 1)]], y=3))
while passengers_on_board != 0:
    for aisle in aisles:
        aisle.move()
    move_in_buffer()
    sec += 1
    print(sec, passengers_on_board)
```