Разработка веб-сайта для нахождения оптимальной рассадки учеников в классе

Автор — Прокофьев Даниил, ученик 10 «В» класса
МБОУ «Гимназия» г. Обнинска
Руководитель — Утянская Елена Васильевна

Обнинск, 2026

Вступление. Пределы человеческого мозга

У подавляющего большинства классных руководителей рассадку своего класса принято делать вручную. Но эффективно ли это? Учитывает ли такая ручная рассадка предпочтения всех и каждого?

Вряд ли. Американским психологом (Miller 1956) Джорджем Миллером было выяснено, что человек не способен в своей кратковременной памяти более 7 ± 2 факторов одновременно.

Это значит, что оптимальность рассадки, создаваемой учителем снижается с увеличением количества учеников и требований к рассадке. Если рассадить до 12 человек с не более чем 5-7 ограничениями оптимально для учителя не составляет никакого труда, то класс на 30 человек с его десятками перекрестными ограничениями и предпочтениями будет рассажен плохо.

Целью данной работы является изучение способов решения задачи рассадки и учеников и написание веб-сайта, который поможет учителям быстро и эффективно рассаживать учащихся.

Формализация задачи

Чтобы определить, является ли решение правильным, нужно придумать, какое решение будет считаться оптимальным.

Было выделено четыре типа ограничений и предпочтений, которые удовлетворят желания большинства пользователей и не ограничивают его свободу выбора:

- Ограничения, связанные с **медицинскими предписаниями** (например, ученик должен сидеть за первой партой согласно предписанию врача)

- Ограничения, связанные с **нежелательным соседством** двух учеников (например, двое учеников враждуют)
- Предпочтения по **рядам и партам** (например, ученик хочет сидеть на первой парте)
- Предпочтения учеников по **соседству** (например, двое друзей хотят сидеть вместе)

Задача относится к классу NP-полных, что легко доказывается сведением ее к задаче о сумке (или ее многомерной версии). Это значит, что не существует алгоритма, который за разумное (полиноминальное) время сумел бы решить задачу. Тем не менее, есть эвристические алгоритмы, которые способны найти приближенный ответ, иногда довольно близкий к наилучшему. Рассмотрим некоторые из них, что применимы в нашем случае

Жадный алгоритм

Для решения задачи можно попробовать использовать алгоритм, который мне кажется самым удобным при ручной рассадке учеников. В чем его суть?

- Выставим учеников в очередь. Сначала поставим тех, кто имеет медицинские, потом тех, кто имеет просто предпочтения по ряду или парте, потом тех, кому вместе сидеть нельзя и, наконец друзей (порядок предпочтений можно менять)
- Возьмем одного ученика из очереди и посадим его на лучшее в данный момент для него место (принцип жадности)

Однако, например, посаженный на первых порах неудачно, ученик будет мешать возникновению оптимальной рассадки. Алгоритм может и найдет какое-то решение, но это будет локальный максимум, а не глобальный, который мы ищем.

Метод отжига

Процесс отжига в физике — это когда металл сначала сильно нагревают, а потом медленно снижают его температуру. Идея в том, чтобы выстроить атомы в кристаллическую решётку и сформировать как можно больше таких устойчивых решёток в металле. Если всё сделать правильно, у металла снижается твёрдость, убираются внутренние напряжения и металл становится более однородным — так его легче обрабатывать.

Если выписать из процесса отжига ключевые мысли и точки, то получим интересное:

- есть система, которая находится не в нужном для нас состоянии

- есть процессы, которые могут протекать в этой системе
- есть параметр (температура), благодаря которому система сама может отрегулировать себя и прийти в нужное состояние
- и есть конечное значение (температуры), по достижении которого система считается максимально близкой к нужному состоянию

Однако сам алгоритм довольно медленный, так как для получения оптимального результата требует медленного охлаждения. А при быстром охлаждении алгоритм все так же может застревать в локальном максимуме.

Генетический алгоритм

Обратимся к другому алгоритму - генетическому. Будем развивать не одно решение, а целую популяцию, и будем производить над ней такие операции как естественный отбор, скрещивание, случайная мутация.

Однако в данном случае эволюция получается слепая. То есть мутации часты, но случайны, и на то, чтобы переместить двух учеников местами, алгоритму может потребоваться очень много поколений, что увеличивает вычислительную сложность.

Меметический алгоритм

Генетический алгоритм можно усовершенствовать. Добавим к эволюции механизм “обучения” особи.

Будем к наилучшей особи и к случайной с шансом 5% применять локальный поиск. Локальный поиск будет совершать некое число итераций, пытаясь на каждой из них поменять двух случайных учеников так, чтобы качество рассадки улучшилось. Если после перемещения рассадка улучшилась, ты мы оставляем новую версию, если нет, то возвращаем все, как было.

Этот подход предотвращает застревание в локальной максимуме и помогает в особо тяжелых случаях.

Как хранить рассадки в памяти?

Использовать одномерные массивы, поскольку с ними любые операции производить гораздо легче. Однако возникает проблема - если учеников в классе меньше чем мест, то для того, чтобы ученик мог занять любое место в классе необходимо, чтобы одномерный массив был размера большего, чем количество учеников.

Каждому ученику присваивается номер от 1 до n (количества учеников). Все, что больше n, считается как пустое место.

```
[1, 3, 5, 6, 7, 2, 4, 8, 9, 10] - класс размером 2 ряда * 5 парты
↪  в ряду, но только 8 учеников
```

Можно представить как:

```
 1        3
 5        6
 7        2
 4        8
9(-1)   10(-1)
```

Как работает меметический алгоритм?

Рассмотрим сам цикл популяций.

1. Идем по поколениям и делаем число этих поколений максимально большим, чтобы алгоритм не застрял:

$$Generations = min(300 + 20 \cdot n, 2500)$$

2. Оценим каждую особь с помощью функции fitness()
3. Найдем индекс лучшей рассадки в scores
4. Проверяем для предотвращения стагнации - если за последние поколение результат не улучшился более чем на 0.001, прибавляем 1 к stagnationCounter
5. Если stagnationCounter >= stagnationLimit (максимального количества поколений, в течение которых позволено оценке лучшей рассадки не меняться достаточно сильно в лучшую сторону), то прерываем программу.
6. Встряхиваем популяцию, если алгоритм близок к стагнации
7. Копируем лучшую особь в новую популяцию и применяем к ней локальный поиск (элитизм)
8. С помощью турнира отбираем две случайные особи и их скрещиваем
9. С небольшим шансом в получившемся потомке меняем двух учеников случайным образом

Код функции, запускающей алгоритм решения без вспомогательных подпрограмм представлен в Приложении 1.

Как реализована проверка предпочтений?

Для проверки предпочтений были написаны вспомогательные функции: `checkMed`, `checkPref`, `checkFriends` и `checkEnemies`. Все они возвращают значение от 0.0 до 1.0 - насколько выполнено каждое из предпочтений ученика.

Рассмотрим одну из них:

```
func checkPref(student optStudent, row, col int) float64 {
    score := 0.0
    maxPossible := 0.0
    if len(student.pCols) > 0 {
        maxPossible += 1.0
    }
    if len(student.pRows) > 0 {
        maxPossible += 1.0
    }
    if maxPossible == 0 {
        return 1.0
    }
    if student.pCols[col] {
        score += 1.0
    }
    if student.pRows[row] {
        score += 1.0
    }
    return score / maxPossible
}
```

Видно, что функция считает сначала максимальное количество возможных баллов за выбранные ряд и парту (нужно, так как не у каждого могут быть такого вида предпочтения) и фактическое количество баллов за место. Затем делит одно на другое и получает число от 0 до 1. Это и будет учтенность предпочтения.

Стоит сказать, что для ускорения работы программы был использован прием создания статического массива. Взяв предпочтения, которые не зависят от учеников рядом, мы для каждого построили массив, который для каждого возможного места будет содержать количество баллов за место (учитывая предпочтения по месту и медицинские ограничения). Это намного быстрее, чем каждое поколение для каждого ученика проходить по его массиву парт или рядов и искать, есть ли там такое значение или нет.

Для проверки друзей и нежелательного соседства используется матрица смежности. Они содержит записи вида:

```
friendsMap[id1][id2] = true // если они хотят быть друзьями
```

где `id1` и `id2` являются номерами учеников

Функция fitness(): как оценить рассадку?

Оценка рассадки есть сумма удовлетворенности каждого из учеников:

```
func fitness(seating []int, config ClassConfig, w Weights,
 ↪  friends SocialMap, enemies SocialMap, staticScores
 ↪  []float64, nStudents int, friendsCount []int) float64 {
    score := 0.0
    for i, studentIdx := range seating {
        if studentIdx < 0 || studentIdx >= nStudents {
            continue
        }
        row, col := i/config.Columns, i%config.Columns
        fScore := checkFriends(studentIdx, seating, row, col,
 ↪  config, friends, nStudents, friendsCount)
        ePenalty := checkEnemies(studentIdx, seating, row, col,
 ↪  config, enemies, nStudents)

        sScore := (fScore * w.FriendBonus * 100.0) - (ePenalty *
 ↪  w.EnemyPenalty * 5.0 * 100.0)
        sScore +=
 ↪  staticScores[studentIdx*config.Rows*config.Columns+i]

        score += sScore
    }
    return score
}
```

Для ускорения программы функции оценки для каждой рассадки в поколении запускается параллельно друг с другом, что позволяет существенно снизить время выполнения на многопоточных и многоядерных процессорах.

Веса фитнес-функции

Каждое значение функции проверки предпочтений домнажается на число - вес данного предпочтения. Это число, которое получено от пользователя, лежит в промежутке от 0.0 до 1.0 и показывает, насколько одно из предпочтений/ограничений важнее другого. Их 4 + пятое, которое отвечает за то, насколько сильно алгоритм будет пытаться сажать учеников ближе к доске.

Вот как это выглядит в запросе к серверу:

```
...
"PriorityWeights": {
          "Medical": 0.9,
          "Friends": 0.65,
          "Enemies": 0.7,
          "Preferences": 0.6,
          "Fill": 0.3
      }
...
```

Выбор языка программирования для написания алгоритма

Для написания самого алгоритма был использован язык программирования Go. Он обладает такими свойствами, как быстрая скорость выполнения (вследствие того, что язык компилируемый и имеет строгую типизацию), поддержка параллельности (через goroutine), возможность развернуть программу, которая сразу превращается в исполняемый файл легко и быстро на почти любом компьютере.

Архитектура

Стоит отметить, что “вычислитель” и сам веб-сайт разделены между собой. Это сделано для того, чтобы компьютер пользователя не перегружался вычислениями, что немаловажно, учитывая, какие маломощные компьютеры могут быть предоставлены учителю школой

Можно сам проект разделить на три части:

- бекенд(сервер) - отвечает за выполнение меметического алгоритма, получает входные данные от фронтенда
- фронтенд(веб-сайт) - получает входные данные от пользователя, передает их бекенду
- CI/CD - то, что отвечает за сборку Docker контейнеров бекенда и фронтенда через GitHub Actions

Интерфейс пользователя

Стояла задача разработать понятный и удобный интерфейс. Нужно было отделить возможность сохранять классы, выбирать их в генераторе и изменять их параметры друг от друга, а также экспортировать файл рассадки в PDf, сохранять получившиеся рассадки и классы в памяти.

Для этого был использован фреймворк Vue.js и библиотека bootstrap-vue-next (специальная версия Bootstrap, написанная для лучшей интеграции в Vue.js) и множество других пакетов, список которых можно посмотреть в приложении 2, содержащем файл package.json (в проектах на Node JS файл, где зафиксированы все пакеты, необходимые для сборки проекта)

Рис. 1. Главный экран приложения

Структура файлов

Главными файлами здесь являются те, что заканчиваются на .Vue - в Vue.js это называется компонентами - составные кусочки сайта - где-то целые страница, где-то части интерфейса.

- Файлы в папке composables/ содержат всю логику фронтенда
- ClassEditor.vue - редактор класса

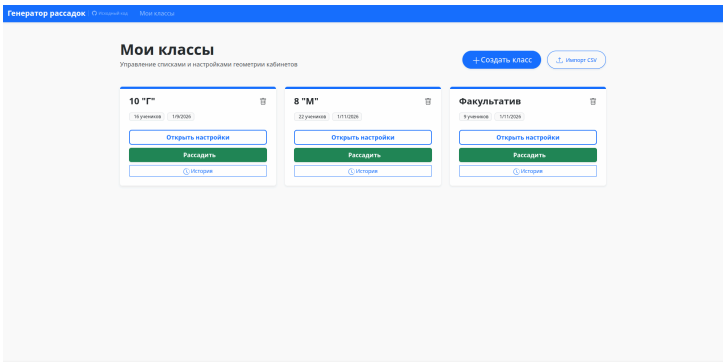

Figure 1: Рис. 1. Главный экран приложения

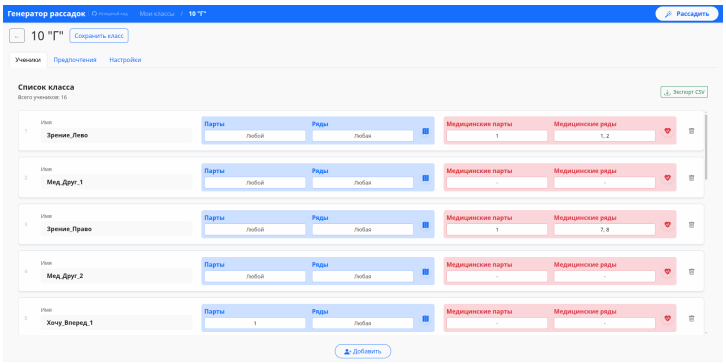

Figure 2: Рис. 2. Интерфейс редактора класса

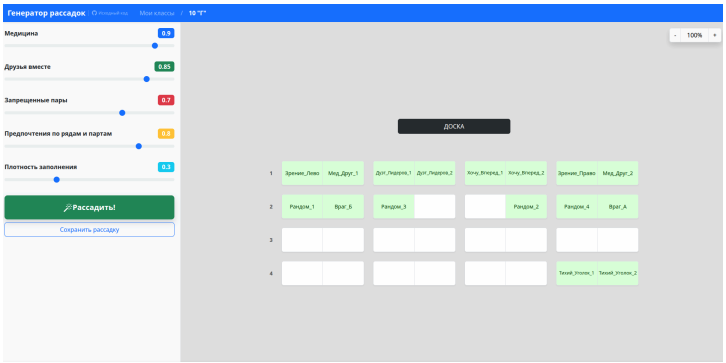

Figure 3: Рис. 3. Параметры генератора рассадки

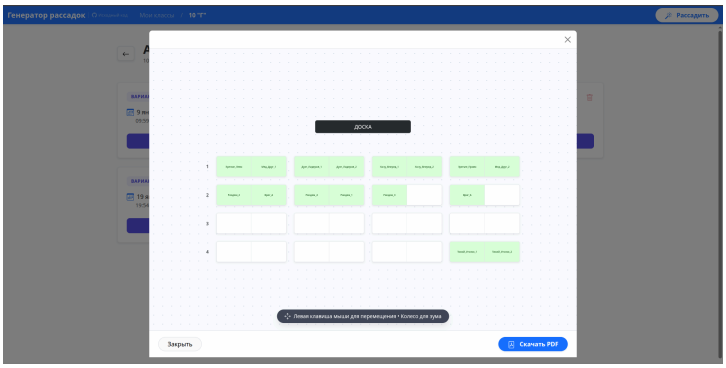

Figure 4: Рис. 4. Страница просмотра истории рассадок

- ClassesList.vue - список классов с главного экрана
- ClassMap.vue - визуализатор рядов и парт, вынесен в отдельный компонент из-за последующего переиспользования
- Generator.vue - страница самого генератора, что отправляет запросы к бекенду
- SeatingHistory.vue - страница, где можно посмотреть все сохраненные пользователем рассадки.

Стоит сказать, что все классы и рассадки сохраняются в localStorage браузера пользователя. Это позволяет пользователю не терять свои данные при перезагрузке страницы и не только. Если целенаправленно не стирать историю браузера, выбирая удаление кеша и cookie, то все будет надежно хранится.

Автоматическая сборка

Для удобства была настроена автоматическая сборка Docker контейнеров фронтенда и бекенда через GitHub Actions. При любом изменении репозитория запускается процедура сборки Docker-контейнеров.

Пример: файл-workflow для сборки бекенда

```
name: Build and Push Go Backend Engine Docker Image

on:
  push:
    branches: [ "main", "master", "dev" ]
    tags: [ 'v*.*.*' ]

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ${{ github.repository }}
```

```
jobs:
  build-and-push:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Log in to the Container registry
        uses: docker/login-action@v3
        with:
          registry: ${{ env.REGISTRY }}
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Extract metadata (tags, labels) for Docker
        id: meta
        uses: docker/metadata-action@v5
        with:
          images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}
          tags: |
            type=raw,value=latest,enable=${{ github.ref ==
↪  'refs/heads/master' }}
            type=sha,format=short

      - name: Build and push Docker image
        uses: docker/build-push-action@v5
        with:
          context: .
          push: true
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}
```

Веб-сайт в Интернете

Благодаря автоматической сборке несложно было и самому развернуть генератор, что я и сделал. Теперь на домене seating-generator.ru находится тот самый веб-сайт, который я разрабатывал. Каждый желающий может получить к нему доступ.

Лицензирование

Проект полностью лицензирован под GNU Affero Public License v. 3. Это значит:

- любой пользователь всегда будет иметь свободу изменять, распространять, неограниченно устанавливать, развертывать, использовать и изучать программу
- любое изменение к программе должно быть выпущено с той же самой лицензией, что гарантирует свободность программы
- если сервис доступен через Интернет, исходный код обязан быть открытым (требования версии Affero)
- пользователь должен иметь право знать, как обрабатываются его данные

А из этого непосредственно следует:

- любой может улучшить генератор
- любой может развернуть свою собственную копию для школы или университета

Ознакомиться с проектом

Ознакомиться с проектом можно по ссылке. Ссылка ведет на мета-репозиторий с презентацией, речью и данным описанием, а также ссылками на репозитории фронтенда и бекенда.

Приложение 1. Код функции RunGA - функции, осуществляющей меметический алгоритм

```
func RunGA(req Request) ([]Response, float64, int) {
    N := req.ClassConfig.Columns * req.ClassConfig.Rows
    nStudents := len(req.Students)
    gaConfig := calcGAConfig(nStudents)
    popSize, generations := gaConfig.PopSize,
↪   gaConfig.Generations
    weights := calculateWeights(req.PriorityWeights)
    numCPU := runtime.NumCPU()

    idToIndex := make(map[int]int)
    opt := make([]optStudent, nStudents)
    for i, s := range req.Students {
        idToIndex[s.ID] = i
        m := func(sl []int) map[int]bool {
            r := make(map[int]bool)
            for _, v := range sl {
```

```
                r[v] = true
            }
            return r
        }
        opt[i] = optStudent{
            Student: s, index: i,
            pCols: m(s.PreferredColumns), pRows:
↪   m(s.PreferredRows),
            mCols: m(s.MedicalPreferredColumns), mRows:
↪   m(s.MedicalPreferredRows),
        }
    }

    staticScores := make([]float64, nStudents*N)
    for i := 0; i < nStudents; i++ {
        for seatIdx := 0; seatIdx < N; seatIdx++ {
            r, c := seatIdx/req.ClassConfig.Columns,
↪   seatIdx%req.ClassConfig.Columns
            mScore := checkMed(opt[i], r, c)
            pScore := checkPref(opt[i], r, c)
            rScore := scorePosition(r, req.ClassConfig.Rows,
↪   weights.RowBonus)

            val := (pScore * weights.PrefBonus) + (rScore *
↪   weights.RowBonus)
            if mScore > 0 {
                val += mScore * weights.MedPenalty
            } else if mScore < 0 {
                val -= weights.MedPenalty * 20.0
            }
            staticScores[i*N+seatIdx] = val * 100.0
        }
    }

    rands := make([]*rand.Rand, numCPU)
    for i := 0; i < numCPU; i++ {
        rands[i] = rand.New(rand.NewSource(time.Now().UnixNano()
↪   + int64(i)))
    }

    population := make([][]int, popSize)
    for i := range population {
        population[i] = rands[0].Perm(N)
    }
    friends, enemies, friendsCount := buildSocialMap(req,
↪   idToIndex)

    newPop := make([][]int, popSize)
```

```
    for i := range newPop {
        newPop[i] = make([]int, N)
    }
    usedBufs := make([][]bool, popSize)
    for i := range usedBufs {
        usedBufs[i] = make([]bool, N)
    }

    scores := make([]float64, popSize)
    var wg sync.WaitGroup

    bestFitnessEver := -math.MaxFloat64
    stagnationCounter := 0
    stagnationLimit := 200 // Stop if no improvement for 150
↪   generations
    totalGens := 1
    if nStudents < 20 {
        stagnationLimit = 80
    } // Faster stop for small classes

    for gen := 0; gen < generations; gen++ {
        chunkSize := (popSize + numCPU - 1) / numCPU
        for w := 0; w < numCPU; w++ {
            start, end := w*chunkSize, (w+1)*chunkSize
            if start >= popSize {
                break
            }
            if end > popSize {
                end = popSize
            }
            wg.Add(1)
            go func(s, e int) {
                defer wg.Done()
                for i := s; i < e; i++ {
                    scores[i] = fitness(population[i],
↪   req.ClassConfig, weights, friends, enemies, staticScores,
↪   nStudents, friendsCount)
                }
            }(start, end)
        }
        wg.Wait()

        iBest := 0
        for i := 1; i < popSize; i++ {
            if scores[i] > scores[iBest] {
                iBest = i
            }
        }
```

```
        if scores[iBest] > (bestFitnessEver + 0.001) {
            bestFitnessEver = scores[iBest]
            stagnationCounter = 0
        } else {
            stagnationCounter++
        }

        if stagnationCounter >= stagnationLimit {
            // Early exit if results stopped improving
            break
        }

        // Adaptive Mutation Rate: Increase if we are stuck
        mutationRate := 0.15
        if stagnationCounter > 50 {
            mutationRate = 0.4 // "Shake" the population to
↪   escape local optima
        }

        copy(newPop[0], population[iBest])
        localSearch(rands[0], newPop[0], req.ClassConfig,
↪   weights, friends, enemies, staticScores, nStudents,
↪   friendsCount, opt)

        for w := 0; w < numCPU; w++ {
            start, end := w*chunkSize, (w+1)*chunkSize
            if start == 0 {
                start = 1
            }
            if start >= popSize {
                break
            }
            if end > popSize {
                end = popSize
            }
            wg.Add(1)
            go func(s, e int, r *rand.Rand) {
                defer wg.Done()
                for i := s; i < e; i++ {
                    p1Idx := tournamentSelection(r, scores, 3)
                    p2Idx := tournamentSelection(r, scores, 3)
                    CrossOver(r, population[p1Idx],
↪   population[p2Idx], newPop[i], usedBufs[i])
                    if r.Float64() < mutationRate {
                        SwapMutation(r, newPop[i])
                    }
                }
```

```
            }(start, end, rands[w])
        }
        wg.Wait()

        population, newPop = newPop, population
        totalGens++
    }

    bestIdx := 0
    bestAns := fitness(population[0], req.ClassConfig, weights,
↪   friends, enemies, staticScores, nStudents, friendsCount)
    for i := 1; i < popSize; i++ {
        Ans := fitness(population[i], req.ClassConfig, weights,
↪   friends, enemies, staticScores, nStudents, friendsCount)
        if Ans > bestAns {
            bestAns = Ans
            bestIdx = i
        }
    }

    bestIndices := population[bestIdx]
    response := make([]Response, N)
    for i, studentIdx := range bestIndices {
        row, col := i/req.ClassConfig.Columns,
↪   i%req.ClassConfig.Columns
        if studentIdx >= nStudents || studentIdx < 0 {
            response[i] = Response{SeatID: i, Row: row, Column:
↪   col, Student: "-", StudentID: -1}
            continue
        }
        response[i] = Response{
            SeatID: i, Row: row, Column: col,
            Student: opt[studentIdx].Name, StudentID:
↪   opt[studentIdx].ID,
            Satisfaction: getSatisfactionDetails(bestIndices,
↪   row, col, studentIdx, weights, req.ClassConfig, friends,
↪   enemies, friendsCount, opt),
        }
    }
    return response, bestAns, totalGens
}
```

Приложение 2. Файл package.json, содержащий все пакеты, необходимые для сборки фронтенда

```
{
  "name": "frontend",
```

```
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "@popperjs/core": "^2.11.8",
    "axios": "^1.13.2", // для отправки запросов к бекенду
    "bootstrap": "^5.3.8", // интерфейс
    "bootstrap-vue-next": "^0.40.9", // интерфейс
    "jspdf": "^4.0.0", // генерация PDF рассадки
    "jspdf-autotable": "^5.0.7",
    "papaparse": "^5.5.3", // импорт/экспорт CSV файлов для
    ↪  классов
    "unplugin-icons": "^22.5.0", // иконки
    "vue": "^3.5.26", // фреймворк
    "vue-router": "^4.6.4", // роутер для многостраничности
    "vuedraggable": "^4.1.0"
  },
  "devDependencies": {
    "@iconify-json/bi": "^1.2.7",
    "@vitejs/plugin-vue": "^6.0.3",
    "unplugin-vue-components": "^29.2.0",
    "vite": "^7.3.1"
  }
}
```

Список используемых источников

Miller, George A. 1956. “The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information.” *Psychological Review* 63 (2): 81.