Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:

Метод распараллеливания алгоритма замощения по изображению деталей

Студент ИУ7-85Б _____ Т.М.Оберган
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель _____ Ю.В.Строганов
(Подпись, дата) (И.О.Фамилия)

Нормоконтролер _____ _____
(Подпись, дата) (И.О.Фамилия)

2021 г.

РЕФЕРАТ

Расчетно-пояснительная записка 65 страниц, 26 рисунков, 17 листингов, 26 источников, 3 приложения.

Объектом разработки является метод распараллеливания алгоритма замощения.

Цель работы — разработать и реализовать метод замощения поверхности, предложить способ его ускорения за счет параллельных вычислений.

Для достижения поставленной цели необходимо решить следующие задачи:

- анализ способов решения задач замощения;
- анализ подходов к ускорению;
- разработка метода решения задачи о замощении;
- разработка метода распараллеливания алгоритма замощения;
- реализация разработанных методов.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В данной научно-исследовательской работе должны быть проанализированы способы повышения эффективности производственных процессов для минимизации отходов за счет использования компактно упакованных выкроек изделий.

Сведение к минимуму отходов может помочь защитить окружающую среду и приносит различные экономические выгоды: такой подход позволяет достичь большего выхода продукта на единицу входящего сырья, а более эффективное использование продуктов означает снижение затрат на приобретение новых материалов, улучшающее финансовые показатели компании. [1]

Задача создания выкроек наиболее актуальна для больших предприятий в сферах пошива одежды, строительства, конструирования, легкой и лесной промышленности, имеет вариации и нюансы в зависимости от используемых материалов, размеров и формы деталей выкройки, желаемого результата. [2]

1 Аналитический раздел

В данном разделе будет формализована задача, рассмотрены вариации задач генерации раскроя, проанализированы аналоги и способы достижения поставленной цели.

1.1 Вариации задач о замощении

Под задачами раскроя и упаковки понимается широкий класс задач, объединенных однообразной логической структурой и допускающих различное толкование. Логической основой для отнесения какой-либо проблемы к данному классу задач, является наличие двух групп сущностей: областей и объектов. В рамках решения поставленной задачи, устанавливается соответствие элементов этих групп [3]. Как правило одной области соответствует несколько объектов, для каждого из которых фиксировано расположение внутри области. Предполагается, что среди объектов существует такой, что ему может быть назначена хотя бы одна область (разрешимость задачи) [3].

В отечественной и зарубежной литературе этот класс задач встречается под следующими названиями:

- задача раскроя запаса материала;
- задача плотного размещения геометрических объектов в заданной области;
- задача загрузки рюкзака – у области фиксированная вместимость, у объектов есть вес и ценность, необходимо достигнуть максимальной ценности, соблюдая ограничение на вместимость [4];
- задача упаковки ящиков (контейнеров) – упаковка объектов предопределенной формы в конечное число контейнеров предопределенной формы [5];
- задача загрузки транспорта – требуется найти такой вариант загрузки транспорта, при котором стоимость перевозимого груза была бы максимальной [6];

- задача обеспечения ритмичности производственного процесса – требуется достигнуть выполнения равных объемов работ за равные интервалы времени для достижения оптимальной загрузки оборудования [7].

1.2 Анализ существующих программных решений

Эвристики для одномерной и двумерной упаковки

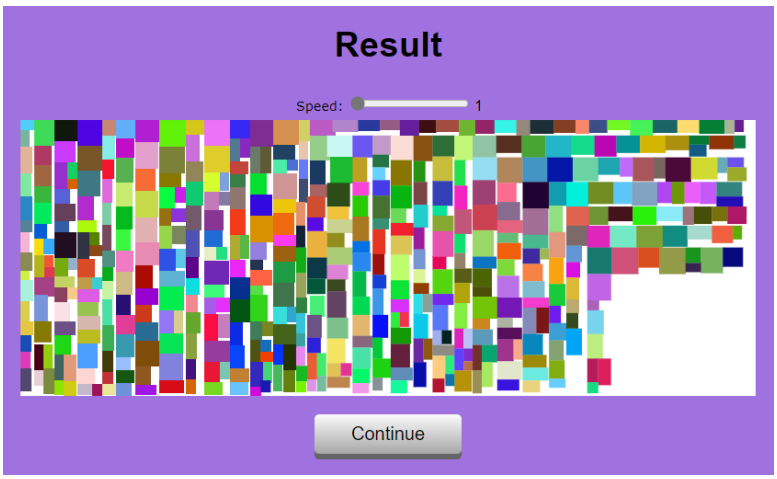

Рис 1.2.1 - интерфейс приложения

Веб приложение для решения двумерной задачи об упаковке прямоугольных объектов в прямоугольные контейнеры.

Достоинства:

- решение как одномерной так двумерной задач;
- возможность настройки работы алгоритма (сортировка по периметру или площади, выбор эвристического алгоритма: первый подходящий, лучший, худший);
- пошаговая визуализация работы алгоритма.

Недостатки:

- не поддерживает объекты произвольной формы;
- не поддерживает контейнеры произвольной формы;
- поддержка только эвристических алгоритмов;

Симплекс-метод и MS Excel

В статье [2] рассматривается методика решения задачи оптимального раскроя при помощи надстройки MS Excel «Поиск решения», использующей симплекс-метод. Описанный подход работает только с прямоугольными объектами и контейнерами.

REDCAFE

Программа Redcafe — это профессиональное программное обеспечение для построения и моделирования выкроек одежды. [8]

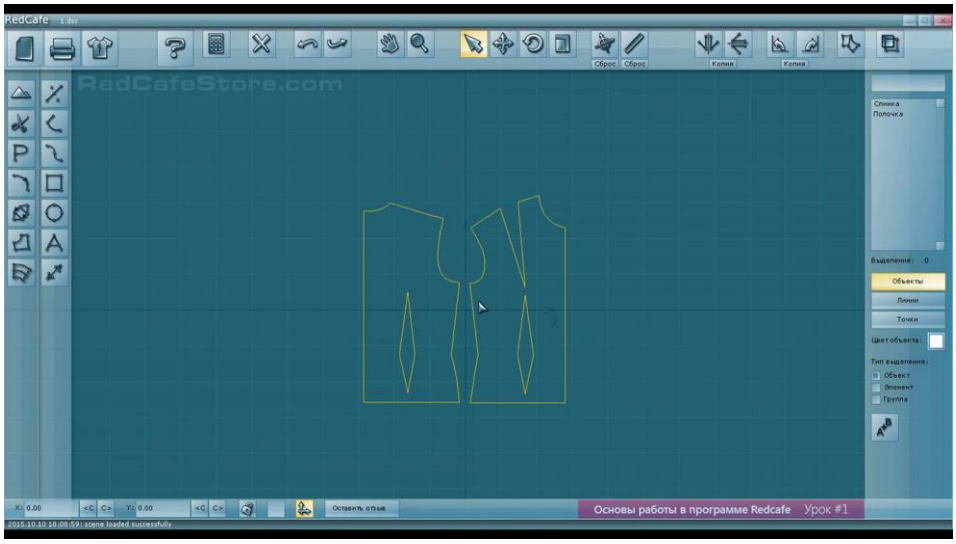

Рис 1.2.2 - интерфейс приложения redcafe

Достоинства:

- объекты могут быть произвольной формы;
- создание новых объектов производится внутри программы.

Недостатки:

- не решается задача генерации оптимального раскроя (пользователь вручную перемещает и вращает детали);
- большая часть функционала программы направлена на швейную сферу.

T-FLEX CAD

Является частью решения T-FLEX PLM в области управления жизненным циклом изделий и организации деятельности предприятий. T-FLEX CAD – это полнофункциональная система автоматизированного проектирования, обладающая средствами автоматизации. [9]

Достоинства:

- имеет готовые модули интеграции с системами AutoCAD, Autodesk Inventor, Creo, Catia;
- объекты могут быть произвольной формы;
- решается задача оптимизации раскроя.

Недостатки:

- программа платная.

1.3 Способы задания плоских объектов

Типовыми методами геометрического моделирования двумерных объектов являются:

- точечный метод – характеризуется набором точек, набором атрибутов – координат точек, отображение, связывающее пару инцидентных точек, является простейшей из моделей;
- математической описание кривой – может быть представлена в параметрической или непараметрической форме:
  - параметрическая форма $\begin{cases} x = f(t) \\ y = g(t) \end{cases}$, где t – параметр,
  - непараметрическая форма описывается в виде явной ($y = f(x)$) или неявной ($f(x, y) = 0$) функции,

  характеризуется узловыми точками, координатами узловых точек и коэффициентами общих уравнений ориентированных кривых, отображением, связывающим пару инцидентных узловых точек;
- метод плоских элементов – характеризуется координатами и углами, задающими положение объекта, параметрами объекта, преобразованием на плоскости, геометрическими операциями объединения и пересечения. [10]

1.4 Метод стыковки плоских фигур

Решение задач оптимального размещения фигур можно получить перемещением фигур относительно друг друга, должны выполняться определенные задачей (непересечение объектов и другие специальные) условия [11].

Движение фигуры может быть поступательным или вращательным. Функцией цели здесь является величина, на которую требуется в заданном направлении или сместить фигуру – при поступательном движении, или повернуть фигуру относительно некоторой точки плоскости – при вращательном движении, до контакта подвижной фигуры с неподвижной. Данный метод назван автором методом стыковки плоских фигур [11].

1.5 Точные алгоритмы решения

1.5.1 Анализ вариантов

В данном алгоритме предполагается последовательный анализ вариантов расположения объектов на области и последующий выбор оптимального. Этот подход предполагает использование принципов динамического программирования. В рамках данного алгоритма используется описанный выше метод стыковки плоских фигур.

1.5.2 Рулонный принцип

В данном случае длина материала настолько больше его ширины, что при планировании раскроя такого материала можно, по существу, пренебрегать концевым отходом. Раскрой рулона на прямоугольные заготовки совершается следующим образом: прямыми поперечными линиями от рулона отрезаются некоторые полосы, которые затем кроятся на необходимые заготовки. [12]

1.5.3 Случай, когда преобладает несколько видов объектов

Пусть из общего списка объектов несколько объектов нужны в значительно большем количестве, чем остальные. В этом случае допустимо отдельно раскроить преобладающие объекты и оставшиеся объекты, а после

создать смешанный раскрой. В зависимости от нужного количества объектов отдается предпочтение тому или иному раскрою [12].

1.5.4 Получение из одного листа одинаковых заготовок

Данный алгоритм целесообразно использовать в рамках предыдущего случая, при решении задачи раскроя группы преобладающих заготовок.

Рассмотрим этот алгоритм на примере раскроя прямоугольной области на прямоугольные объекты. На рисунке 1.6.1 показано сравнение двух вариантов расположения объектов: вдоль и поперек листа. В данном случае оба эти варианта дают одинаковое количество заготовок (40) на лист.

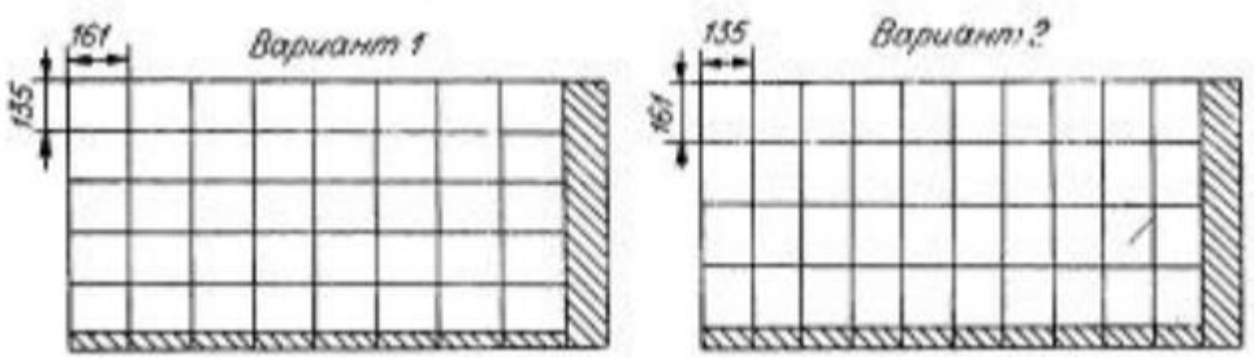

Рис 1.5.2 - сравнение расположения вдоль и поперек листа [12]

Как видно на рисунке 1.6.2, раскрои вдоль и поперек листа не были оптимальными, т.к. из на области аналогичных размеров было размещено 44 заготовки.

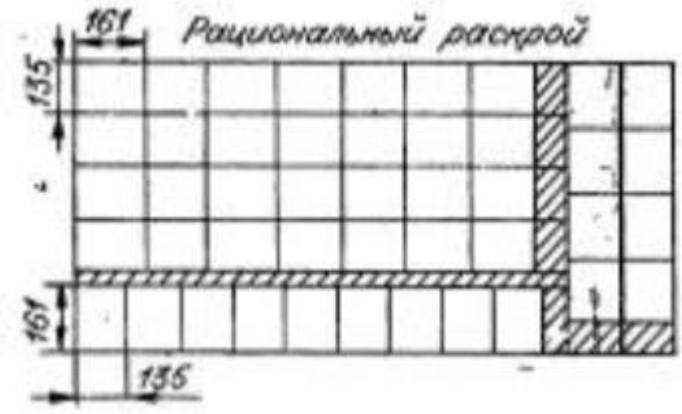

Рис 1.5.3 - рациональный раскрой [12]

1.6 Приближенные алгоритмы

В силу NP-трудности большинства задач упаковки и раскроя известные точные алгоритмы не позволяют решать за приемлемое время задачи, размерность которых соответствует требованиям практических приложений. Альтернативой точным алгоритмам могут служить градиентные аналоги дискретных алгоритмов. Эти алгоритмы дают возможность быстро и с довольно высокой точностью решать задачи большой размерности [13]. Эффективность градиентных алгоритмов заметно увеличивается, если использовать их в виде серий конкурирующих алгоритмов [14].

1.7 Выбор метода расчета

Используемый метод расчета зависит от размеров области и объектов, их количества. Чтобы выбрать метод расчета требуется грубо составить раскрой для длинных объектов. Пусть N_1 – требуемое количество листов для длинных объектов, $N_{1,2}$ – количество листов, которое потребуется для получения длинных и средних объектов, а N – количество листов, при 100% использовании листа всей группой объектов. Тогда можно выделить 4 простейших случая:

1. $\mathrm{N}_1 \geq \mathrm{N}$ – для длинных объектов требуется столько же или больше листов, чем на весь комплект при 100% использовании материала. Данная задача сводится к задаче линейного раскроя.
2. $\mathrm{N}_1 < \mathrm{N}$ – длинные объекты требуют меньшего количества листов, чем весь комплект при 100% использовании листа. Следует приближенно оценить $\mathrm{N}_{1,2}$.
3. $\mathrm{N}_{1,2} < \mathrm{N}$ – в данном случае преобладают мелкие объекты. Используется метод простого перебора.
4. $\mathrm{N}_1 < \mathrm{N}; \mathrm{N}_{1,2} \geq \mathrm{N}$ – случай, требующий детального анализа. Если некоторых объектов требуется значительно больше, то предпочтение отдается методу описанном в разделе 1.6.3, иначе – 1.6.4.

1.8 Существующие подходы к ускорению

1.8.1 MPI

MPI (Message Passing INterface) - это специальный API для облегчения переносимости программы с кластера на кластер. Задача распределения нагрузки на сеть и узлы ложатся на программиста, зависит от особенности строения кластера. [15]

Основными требованиями к MPI являются [16]:

- стандартизация - нет необходимости беспокоится о том какая из реализаций MPI установлена, синтаксис и функциональное поведение соответствуют стандарту;
- переносимость - программы должны иметь возможность выполняться на разных машинах без дополнительных затрат труда по изменению кода, все нюансы по использованию спецификаций аппаратных средств берут на себя функции MPI;
- эффективность работы;
- функциональность - MPI обеспечивает управление буфером сообщения, асинхронную коммуникацию, различные наборы виртуальных топологий, поддержку неоднородных сетей и библиотек.

Для использования MPI рекомендуется предварительно написать и безукоризненно отладить последовательную версию программы. Отладка на этом этапе нужна для экономии сил на отладке параллельной программы, что выполнить сложнее. После этого нужно добавить вызовы функций MPI и удостовериться на небольшом количестве узлов, желательно двух, корректность работы программы. При необходимости, в последствии увеличивать число узлов.

Этапы работы программы, использующей MPI:

1. инициализация;
2. коммуникации распределения данных по процессам;
3. выход из системы передачи сообщений.

Структура сообщений MPI:

- содержимое - отправляемые/получаемые данные
  - буфер - память, выделенная для переменной,
  - начало буфера - адрес начала данных,
  - тип данных;
- сопроводительная информация
  - назначение/источник - равно рангу процесса в коммуникаторе, нужно для определения маршрута сообщения,
  - тег - произвольное число, для различения сообщений,
  - коммуникатор - определяет коммуникационное пространство.

1.8.2 Grid системы

Грид — это система, которая координирует распределенные ресурсы посредством универсальных, открытых, стандартных интерфейсов и протоколов [17]. Обладает следующими качествами:

- распределяет ресурсы (процессорное время, память);
- использует общие интерфейсы и протоколы;
- обеспечивает нужный уровень обслуживания [18].

Главная идея grid систем - совместное пользование ресурсами. Grid ресурс - это компонент системы, который используется для обработки запроса, в качестве вычислительного элемента или хранилища данных [19]. Ввиду широкого распространения персональных компьютеров актуальность грид-систем набирает обороты: в качестве ресурсов можно использовать CPU, процессоры приставок, видеокарт. А высокий уровень технологий в области распараллеливания задач позволяет использовать эти ресурсы для выполнения вычислений аналогично суперкомпьютерам [20].

Грид-технологии реализуют возможность коллективного доступа к общим ресурсам [17]. Для таких систем очень важно обратить внимание на безопасность и надежность коммуникации между элементами системы. Главными требованиями к безопасности в грид являются:

- аутентификация - проверка подлинности сущности в системе (пользователя, процесса, ресурса);
- авторизация - подтверждение прав доступа сущности, в грид-системе владелец ресурса должен иметь возможность управлять доступом к предоставляемому ресурсу;
- конфиденциальность - к информации должен быть доступ только у уполномоченных на это сущностей, особенно важно для систем, обрабатывающих чувствительную информацию.

Классификация грид-систем на основе решаемых задач [21]:

- вычислительный грид - основной целью является достижение максимальной скорости вычислений за счет использования параллельных вычислений на большом числе компьютеров;
- грид для интенсивной обработки данных - нетребователен к вычислительным ресурсам, основная сложность - передача данных;
- семантический грид для оперирования данными из разнотипных БД.

1.8.3 GPU

Графический процессор (GPU) - отдельное устройство, выполняющее графический рендеринг. Благодаря специализированной конвейерной архитектуре он намного эффективнее в обработке графической информации, чем центральный процессор.

GPU вычисления основаны на совместном использовании CPU и GPU: основная часть приложения выполняется на CPU, а требовательная к вычислениям на GPU. За счет высокой производительности (порядка терафлопсов при вычислениях с плавающей запятой) GPU достигается повышенная производительность. [22]

Для программирования под GPU распространены две технологии:

- OpenCL - стандарт, который поддерживают множество производителей видеокарт;
- CUDA - проприетарная технология и SDK от компании NVIDIA

Ограничения при работе с GPU:

- невозможность выделить отдельно одно ядро, выделяется целый блок;
- ядра выполняют одни и те же инструкции, но с разными данными (SIMD - Single Instruction Multiple Data);
- логика в алгоритмах должна быть относительно простой, минимум ветвлений, ввиду простоты логических блоков GPU.

GPU подойдет проектам, если [23]:

- задача приводима к SIMD виду;
- есть возможность закешировать большую часть данных до вычислений на карту;
- задача подразумевает интенсивные вычисления.

1.9 Выводы из аналитического раздела

В данном разделе была формализована задача, рассмотрены вариации задач генерации раскроя, проанализированы аналоги, способы представления плоских объектов, проанализированы точные алгоритмы решения, предоставлена рекомендуемая схема выбора расчета.

Из рассмотренных подходов к ускорению, грид-системы - прекрасный инструмент для решения задач, требовательным к вычислительным ресурсам, за счет гибкости и масштабируемости.

2 Конструкторский раздел

В данном разделе будут рассмотрены требования к методу раскроя, предоставлены схемы алгоритмов и структура приложения, описаны собственные методы решения задачи и ее ускорения.

2.1 Формализация задачи

2.1.1 Требования к методу раскроя

Ниже приведен список требований к методу раскроя.

- Метод должен работать с плоскими объектами произвольной формы.
- Время работы метода конечно.
- Метод должен работать для заполнения прямоугольных областей.
- Метод должен работать с разными размерами объектов, в пределах размеров области.

2.1.2 IDEF0

На рисунках 2.1.1 – 2.1.4 представлена IDEF0 диаграмма.

Рис 2.1.1 - EDEF0 диаграмма ветка A-0

Суть поставленной задачи - разместить заданные изображения по определяемым конфигурацией правилам и отобразить результат в наглядном виде. Она решается в три этапа: считывание данных, поиск размещения, преобразование результата.

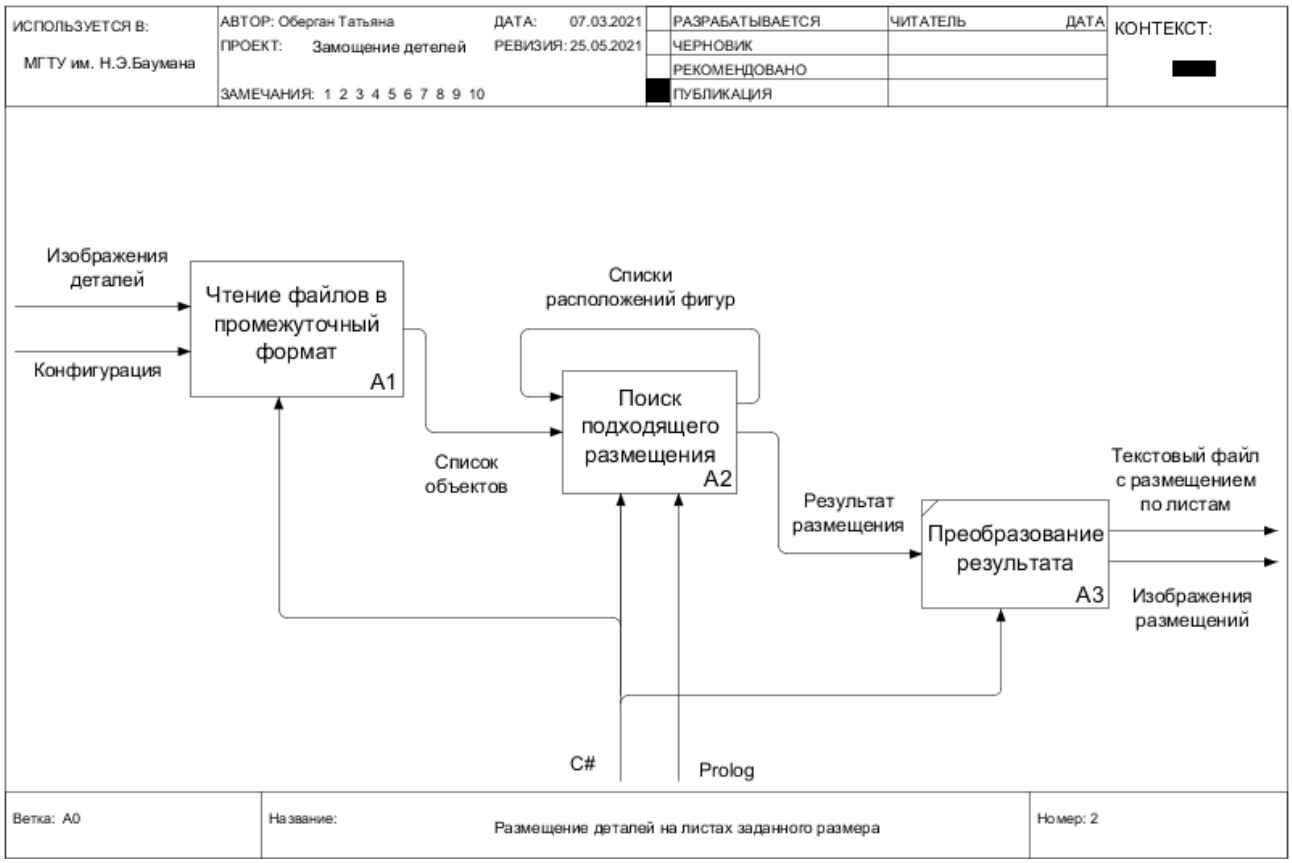

Рис 2.1.2 - EDEF0 диаграмма ветка A0

Настройки конфигурации включают в себя такие поля как:

- размер пространства для размещения (ширина и высота);
- цвет, считываемый с изображения фигур;
- количество деталей;
- шаг поворота фигур;
- коэффициенты масштабирования;
- отступ от границы фигуры;
- путь к директории для хранения промежуточных форм обработки фигур;
- путь к директории для вывода результата.

Этап считывания изображений и конфигурации, предобработка объектов лежит на C#. Этот шаг, как видно на рис. 2.1.3, в свою очередь состоит из трех этапов:

- расширение границ фигур (при необходимости) - нужно, чтобы детали на окончательном раскрое имели между собой запасное расстояние, а не располагались вплотную;
- формирование масштабов объектов - для итеративного поиска результата необходимо отмасштабировать детали для всех заданных в конфигурации размеров;
- формирование поворотов объектов - для учета в поиске всех возможных расположений, необходимо также (наряду с перемещением по ox и oy) учесть вращение фигуры вокруг оси.

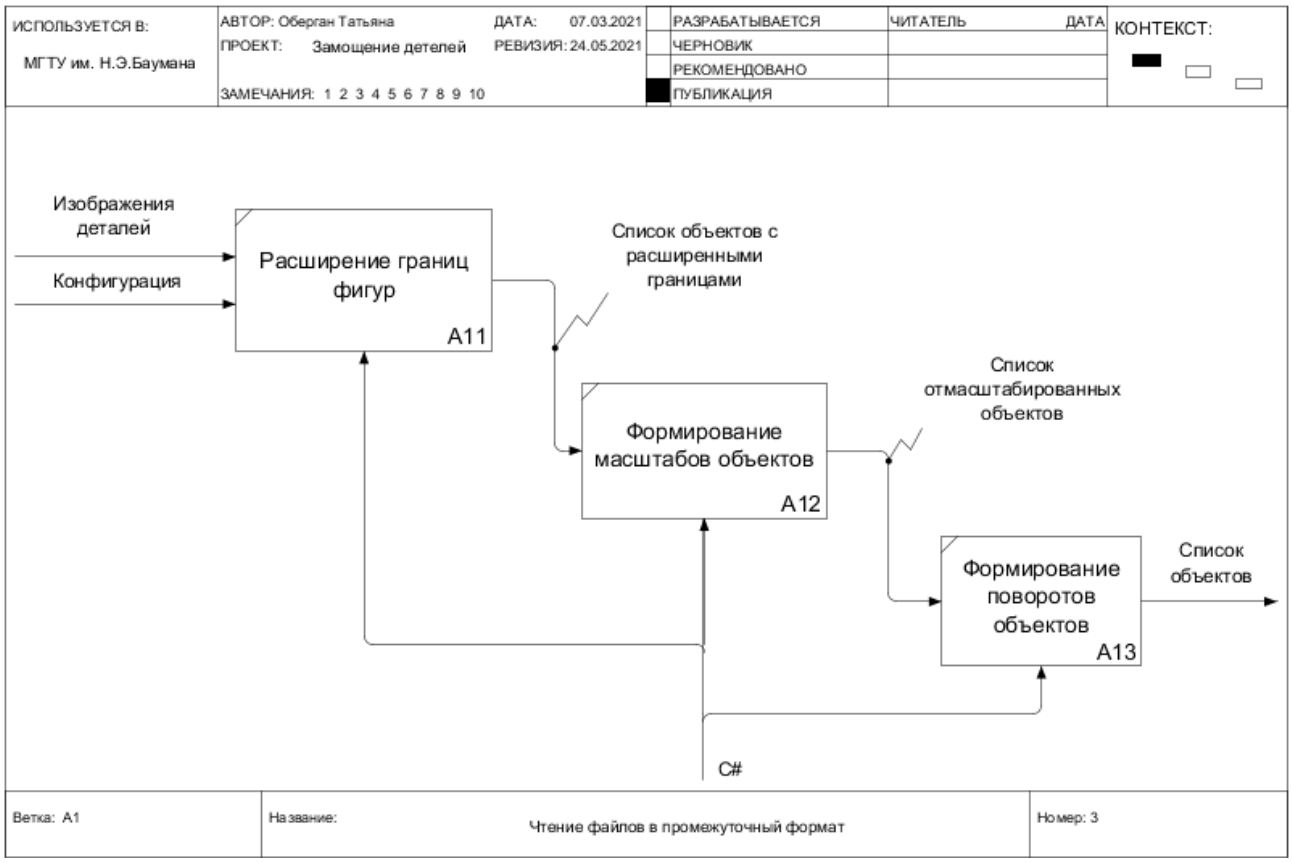

Рис 2.1.3 - EDEF0 диаграмма ветка A1

После прохождения этапа предобработки, изображение детали преобразуется во внутренний формат фигуры, которая представлена списком масштабов, каждый из которых является списком базовых (нужного масштаба) фигур, повернутых вокруг своего центра.

Результаты предобработки направляются модулю поиска подходящего размещения (рис. 2.1.4). Этот модуль итеративно располагает детали на полотне, запоминая подходящие комбинации. Генерация запроса — это прослойка-транслятор между представлением фигур после предобработки и представлением фигур для поиска размещения на листе. В модуле поиска подходящий размещений узким горлышком является обработка запроса ввиду огромного числа комбинаций для перебора на средних и больших размерах деталей.

Проблему узкого горлышка можно решить введением дополнительных мощностей (новых серверов / grid вычислительных ресурсов) и распараллеливанием запросов на несколько серверов.

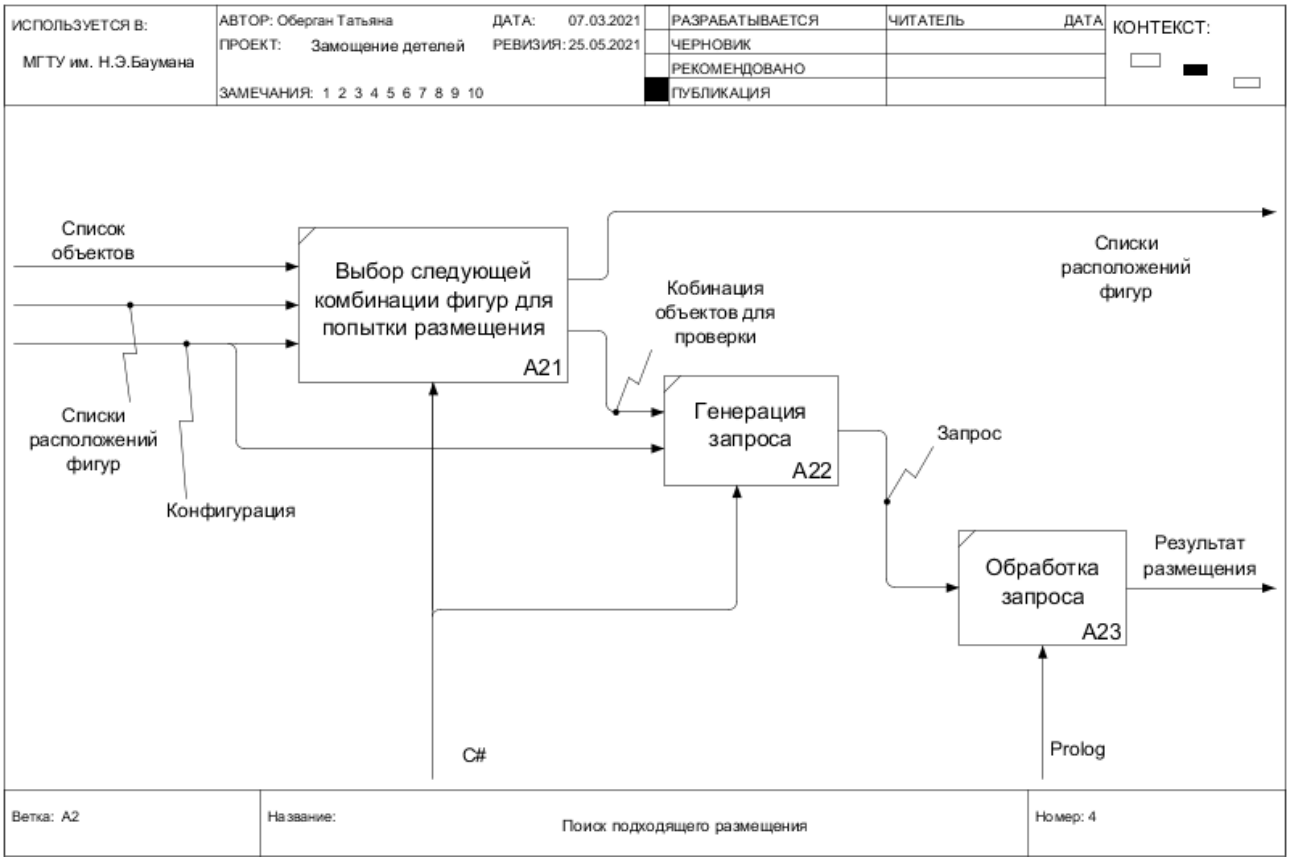

Рис 2.1.4 - EDEF0 диаграмма ветка A2

2.1.3 Use-case диаграмма

На рисунке 2.1.5 продемонстрирована use-case диаграмма. Пользователь системы отдельно запускает вычислительные сервера, а позже добавляет их адреса в систему поиска размещения. Наряду с этим пользователь может задавать изображения деталей для поиска и изменять конфигурацию системы. По мере прогресса поиска размещения, пользователю будет доступна информация о найденных замощениях фигур.

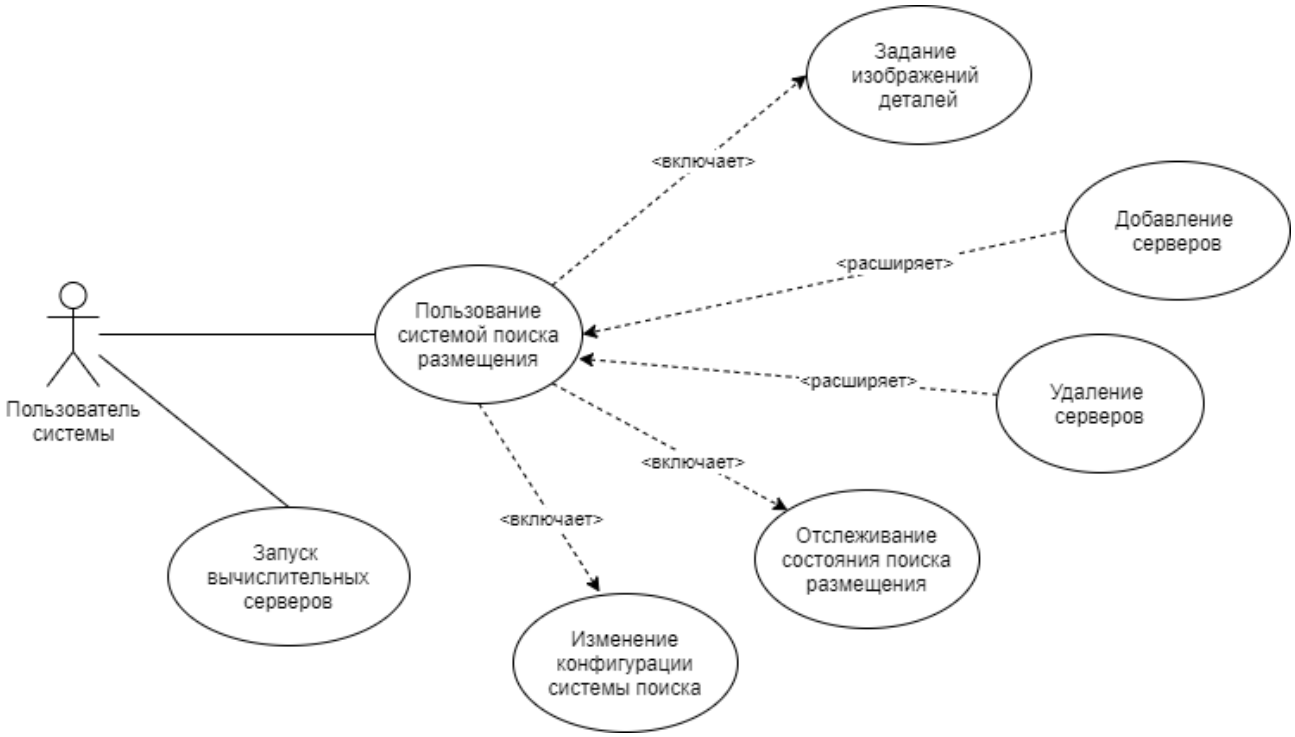

Рис 2.1.5 - use-case диаграмма

2.1.4 Структура приложения

На рисунке 2.1.6 показана структура приложения, из которой наглядно видно, как происходит взаимодействие компонентов: приложение является прослойкой между пользователем и вычислительной группой серверов, менеджер серверов получает запрос и принимает решение какому из серверов его отправить, принимает результаты, следит за исправностью доступа к серверам.

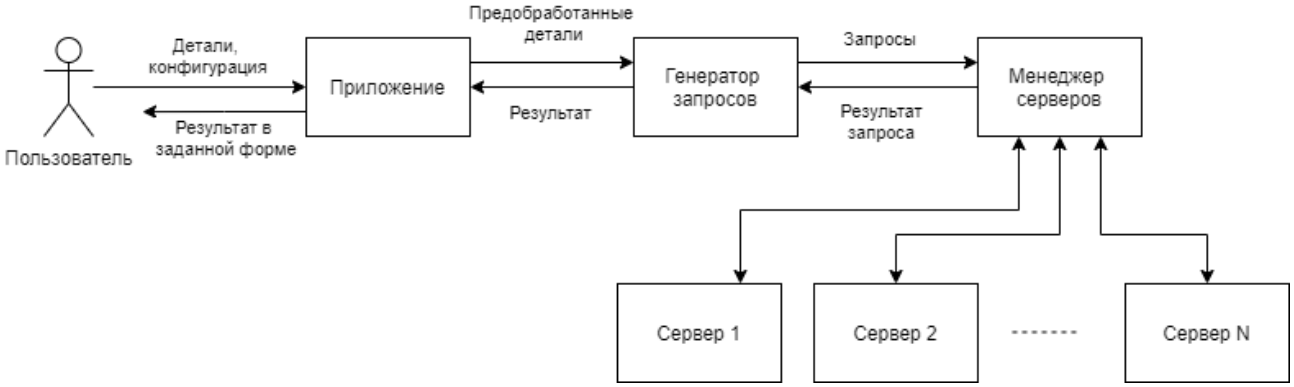

Рис 2.1.6 - структура приложения

На рисунке 2.1.7 показана диаграмма последовательности взаимодействия с сервером для ключевых моментов работы с ним: добавление в список доступных для работы, удаление из этого списка, обновление состояния сервера (в случае если было потеряно соединение) и наиболее важный - отправление запроса на вычисления.

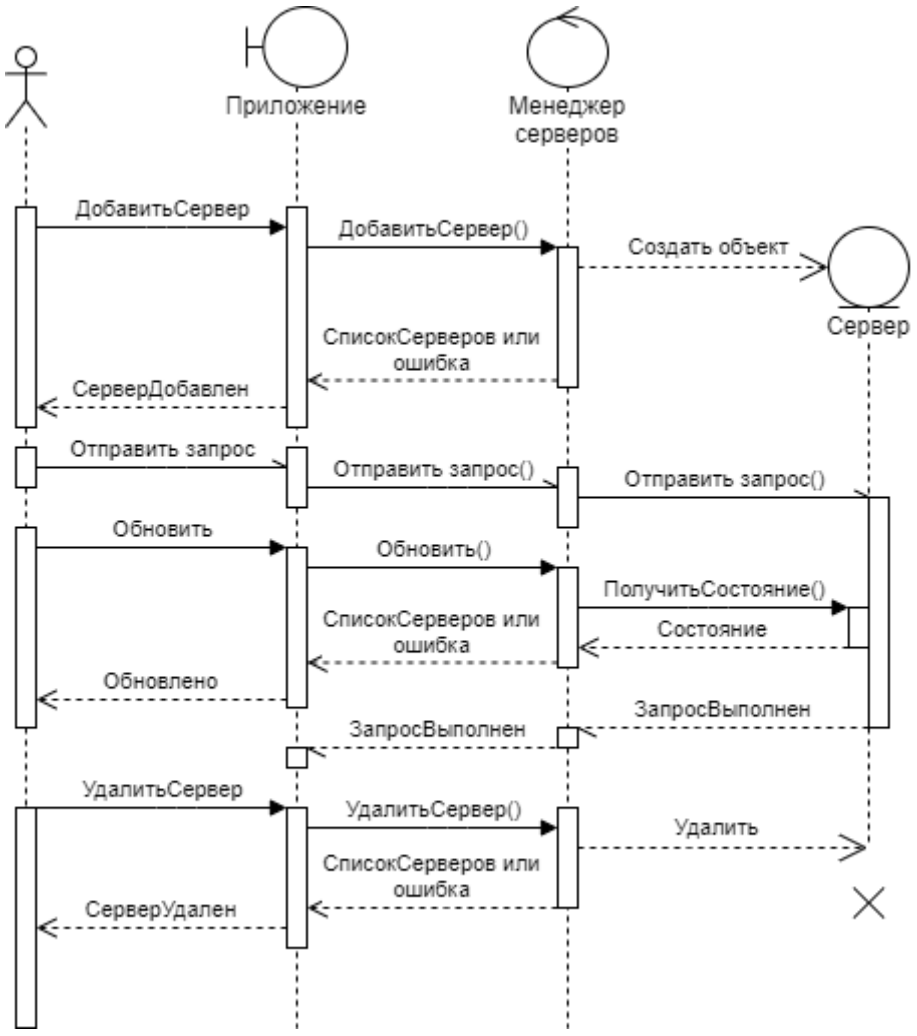

Рис 2.1.7 - диаграмма последовательности взаимодействия с сервером

2.2 Предлагаемый метод

2.2.1 Список объектов

В [14] объекты рекомендуется разделять на три группы:

1. длинные объекты – укладываются только вдоль листа;
2. средние объекты – умещаются как вдоль, так и поперек листа, требуют тщательного анализа;
3. мелкие объекты.

Внутри первой и второй групп объекты упорядочиваются в порядке убывания ширины. В третьей – в порядке убывания суммарной площади группы объектов каждого вида.

2.2.2 Случай, когда преобладает несколько видов объектов

Порядок решения задачи:

1. отдельно решается задача раскроя группы преобладающих заготовок;
2. на основании известного расположения заготовок устанавливается расположение листов и различных полос, остающихся после отрезания от листа заготовок, положение которых уже выбрано;
3. размещаются оставшиеся заготовки;
4. составляется раскройный план с употреблением по возможности полноценных (использование материала листа = 100%) раскроев.

2.2.3 Получение из одного листа одинаковых заготовок

Порядок действий:

1. с помощью округления отношения длины сторон заготовки находятся наиболее благоприятные раскрои короткой стороны листа;
2. в найденной комбинации длин раскраивается значительная часть заготовок, одни ряды заготовок – вдоль большей стороны, другие - поперек;
3. оставшаяся прямоугольная часть листа заполняется из условия наилучшего заполнения ее короткой стороны.

2.2.4 Метод размещения фигур на листе

Метод размещения фигур на листе включает в себя следующие шаги:

1. предобработка фигур, расширение границ, представление в формате отрезков;
2. генерация пространства поиска;
3. перебор позиций и поворотов (проверка помещаются ли фигуры при заданном расположении):
    a. проверка помещаются ли габариты,
    b. проверка помещаются ли центральные сегменты,
    c. попытка вырезать фигуру из списка доступных областей;
4. возврат результата.

2.3 Методы распараллеливания алгоритма замощения

2.3.1 Комбинирование фигур

Одним из способов модификации алгоритма является предварительная обработка некоторых групп фигур. Формируются группы, запрос на размещение из которой отправляется своему серверу. Таким образом кластер серверов параллельно обрабатывает разные группы фигур, для поиска размещений. При нахождении оптимального размещения группы, она объединяется в одну фигуру и в дальнейших расчетах является единым целым.

Метод комбинирования состоит из следующих шагов:

1. предобработка фигур;
2. выделение групп фигур с учетом размеров;
3. поиск оптимального объединения группы фигур;
4. объединение фигур в единый объект;
5. дальнейший поиск оптимального размещения.

2.3.2 Разделение на области

Данный способ модификации заключается в распараллеливании самого метода поиска. Вместо предоставления полного пространства для поиска размещения фигуры, свободное пространство разделяется на области. Один

запрос - поиск в одной области. За счет уменьшения пространства поиска, количество вариантов размещения также уменьшается. Это позволяет скорее найти результат.

Метод разделения на области состоит из следующих шагов:

1. предобработка фигур;
2. цикл поиска размещения:
    a. разделение свободного пространства на области;
    b. для каждой из областей формирование и отправка запроса;
    c. обработка и сохранение результатов;
3. сохранение результатов.

Данный метод стоит использовать или в качестве запасного пути в случае, если основная ветка решения не подойдет, или как ускорение поиска (целесообразно, когда размещение не может быть найдено за четверть времени таймаута).

2.3.3 Дополнительный вариант размещения

Случай, когда размещение не найдено за четверть времени таймаута может означать несколько вещей:

1. количество фигур текущей группы слишком велико и подходящее размещение находится слишком далеко в дереве решений;
2. размещение в принципе не может быть найдено при входной комбинации фигур.

В первом случае целесообразно использовать ускорение поиска, описанное в разделе 2.3.2. Во втором же случае стоит, не дожидаясь истечения времени таймаута, начать рассматривать альтернативную комбинацию фигур. Таким образом параллельно будет происходить поиск двух или более комбинаций замощения листа. Когда основное решение не подойдет можно будет использовать запасные.

2.4 Выводы из конструкторского раздела

В данном разделе были рассмотрены требования к методу раскроя, предоставлены IDEF0 схемы, use-case диаграмма, схемы структуры приложения. Описаны собственные методы решения задачи и ее ускорения.

3 Технологический раздел

В данном разделе будут выбраны язык программирования и среда разработки, продемонстрированы собственные реализации и описано взаимодействие с Microsoft Azure.

3.1 Выбор и обоснование языка программирования и среды разработки

В качестве языка программирования был выбран C# т.к.:

- имеется достаточный опыт программирования на этом языке, что сократит время написания программы;
- данный язык программирования кроссплатформенный и объектно-ориентированный, что даст в полной мере использовать наследование, абстрактные классы и т.д.; [24]

В качестве среды разработки была выбрана «Visual Studio 2019» по следующим причинам:

- она бесплатна в пользовании студентами;
- она имеет множество удобств, которые облегчают процесс написания и отладки кода;
- она обеспечивает работу с Windows Forms – интерфейсом, который упрощает доступ к элементам интерфейса Microsoft Windows за счет создания обертки для существующего Win32 API в управляемом коде;
- имеется достаточный опыт программирования при помощи данной среды разработки, что сократит время изучения возможностей.

Для реализации серверной части был использован swi-prolog т.к. это логический язык программирования, решающий задачу полного перебора [25]. В сравнении с другими языками, на прологе ее можно решить с меньшими трудозатратами программиста.

- подходит для решения задач, где рассматриваются объекты и отношения между ними;

- имеет возможность реализации сервера с дальнейшим обращением посредством post и get запросов.

Т.к. ресурсы локальной машины сильно ограничены была использована облачная платформа Microsoft Azure для хостинга серверов. Для ускорения процесса настройки сервера, использован Docker.

3.2 Собственные реализации

В этом разделе будут рассмотрены различные реализованные способы решения задачи размещения деталей: пиксельное представление, оптимизация при помощи опорных точек, y-отрезки, оптимизация масштабированием.

3.2.1 Пиксельное представление

В данной реализации фигура представлена списком поворотов, каждый из которых задается углом и списком дельт.

На вход поступает изображение, считываемое как Bitmap, далее инициализируется класс Figure, который отвечает за хранение всей нужной информации о детали: путь к файлу, название фигуры, изображение, уникальный индекс внутри сессии, шаг поворота, цвет, базовое состояние фигуры без какой-либо предобработки и список повернутых состояний фигуры (с увеличением границ).

Листинг 3.2.1 - поля класса Figure

```
1. public class Figure
2. {
3.     private string path;
4.     public string name;
5.     Bitmap bitmap;
6.     public int id = -1;
7.     int angleStep;
8.     Color figColor;
9.
10.     public DeltaRepresentation noScaling;
11.     public List<DeltaRepresentation> rotated = new
   List<DeltaRepresentation>();
12. }
```

На листинге 3.2.2 показана основная информация, хранящаяся в классе DeltaRepresentation: координаты центра фигуры, угол поворота относительно базового состояния детали, список дельт (пиксельное представление).

Листинг 3.2.2 - поля класса DeltaRepresentation

```
1. public class DeltaRepresentation
2. {
3.     public int xCenter, yCenter;
4.     public double angle = 0;
5.
6.     public List<Point> deltas = new List<Point>();
7. }
```

Каждая дельта является кортежем (x, y) и считается как (x_текущий_пиксель - x_центр; y_текущий_пиксель - y_центр).

Для поиска решения в прологе требуется для начала задать пространство поиска, по условию задачи это прямоугольный лист. На листинге 3.2.3 показана функция, генерирующая код, для задания пространства поиска в прологе.

Листинг 3.2.3 - функция задания пространства

```
1. /// <summary>
2. /// Формирует строку вида [(0,[0,1,2,3]),(1,[0,1,2,3])]
3. /// При x = 4, y = 2
4. /// </summary>
5. public static string CreateLst(int x, int y)
6. {
7.     string res = "[";
8.     string xValues = "[" + String.Join(",",
   Enumerable.Range(0, x).ToArray()) + "]";
9.
10.     for (int yCur = 0; yCur < y - 1; yCur++)
11.     {
12.         res += "(" + yCur + "," + xValues + "),";
13.     }
14.     res += "(" + (y - 1) + "," + xValues + ")]";
15.
16.     return res;
17. }
```

Далее фигуры, считанные ранее также требуется привести в требуемое сервером представление. Фигуры преобразуются в следующий формат:

Фигура = [(угол_1, [(x1, y1), (x2, y2), ...]),

(угол_2, [(x1, y1), (x2, y2), ...]), …]

Сформированные поле и список фигур передаются в предикат place_it3, где происходит его рекурсивный вызов для каждой из фигур в списке. Если центр текущей фигуры - не занятая клетка, происходит попытка «вырезать» деталь из пространства, при успехе происходит попытка подставить следующую фигуру, при неудаче происходит одно из:

1. переход к следующему углу детали;
2. смещение детали;
3. откат к поиску следующего расположения для прошлой детали.

Ниже предоставлены листинги предикатов поиска размещения.

Листинг 3.2.4 - предикат поиска ответа

```
1. place_it3(_,[],_):-fail.
2. place_it3([],_,[]):-true.
3. place_it3([[(Angle,H)|_]|L],F,[(X,Y,Angle)|Ans]):-
4.     mymember(X,Y,F),
5.     delete_it((X,Y,H),F,F2),
6.     place_it3(L,F2,Ans).
7. place_it3([[_|Hs]|L],F,Ans):-
8.     place_it3([Hs|L],F,Ans).
```

Листинг 3.2.5 - предикат установки принадлежности точки пространству

```
1. mymember(X,Y,[(Y,Xs)|_]):- member(X, Xs).
2. mymember(X,Y,[_|L]):- mymember(X,Y, L).
```

Листинг 3.2.6 - предикат вычитания детали из пространства

```
1. delete_it((_,_,[]),F,F):-!.
2. delete_it((X,Y,[(Y1,X1s)|Lst]),[(Y2,X2s)|F],[(Y2,X3s)|F2]):-
3.     Y2 #= Y+Y1,
4.     mymap2(X2s,X1s,X,X1s2),
5.     subtract(X2s, X1s2, X3s),
6.     delete_it((X,Y,Lst),F,F2).
7. delete_it((X,Y,Lst),[XYZ|F],[XYZ|F2]):-
8.     delete_it((X,Y,Lst),F,F2).
```

3.2.2 Опорные точки

В дополнение к пиксельному представлению находятся 4 опорные точки, для каждого угла поворота. Например, на рисунке 3.2.3 это будет [(200, 200), (200, -200), (-200, 200), (-200, -200)]. Это дает преимущество в скорости нахождения подходящего размещения т.к. перед подстановкой фигуры целиком, для начала можно проверить, что опорные точки помещаются.

Опорные точки = [(правая верхняя дельта), (правая нижняя дельта), (левая верхняя дельта), (левая нижняя дельта)]

На листинге 3.2.7 показан метод класса DeltaRepresentation, который в цикле определяет 4 крайние точки текущего объекта. Опорные точки считаются единожды, результат вычислений хранится и при повторном вызове метода GetOuterDots() будет возвращен.

Листинг 3.2.7 - метод поиска опорных точек фигуры

```
1. public SortedDictionary<int, List<int>> GetOuterDots()
2. {
3.     if (specialDots != null)
4.         return specialDots;
5.
6.     Point minX = new Point(int.MaxValue, int.MaxValue);
7.     Point minY = new Point(int.MaxValue, int.MaxValue);
8.     Point maxY = new Point(int.MinValue, int.MinValue);
9.     Point maxX = new Point(int.MinValue, int.MinValue);
10.
11.   foreach (Point p in deltas)
12.     {
13.         if (minX.X > p.X)
14.             minX = p;
15.         if (minY.Y > p.Y)
16.             minY = p;
17.         if (maxX.X < p.X)
18.             maxX = p;
19.         if (maxY.Y < p.Y)
20.             maxY = p;
21.     }
22.
23.     List<Point> outerDots = new List<Point> { minX, minY,
  maxX, maxY };
24.     specialDots = TransformDeltaToDict(outerDots);
25.     return specialDots;
26. }
```

3.2.3 Y-отрезки

Фигура представлена списком поворотов, каждый из которых задается списком отрезков, сгруппированных по y, где yi уникально для одного списка описания одной фигуры, а отрезок является кортежем из двух координат по x: началом и концом отрезка.

Фигура = [(угол_1, [опорные точки1],
[(y1, [отрезки для y1]), (y2, [отрезки для y2]), …]),
(угол_2, [опорные точки1],
[(y1, [отрезки для y1]), (y2, [отрезки для y2]), …]), …
]

Отрезки для yi = [(x_начало, x_конец_1_отрезка), …, (x_начало_n_отрезка, x_конец)]

Например, для рисунка 3.2.3: отрезки для y200 = [(-200, 200)], отрезки для y0 = [(-200, -100), (100, 200)]

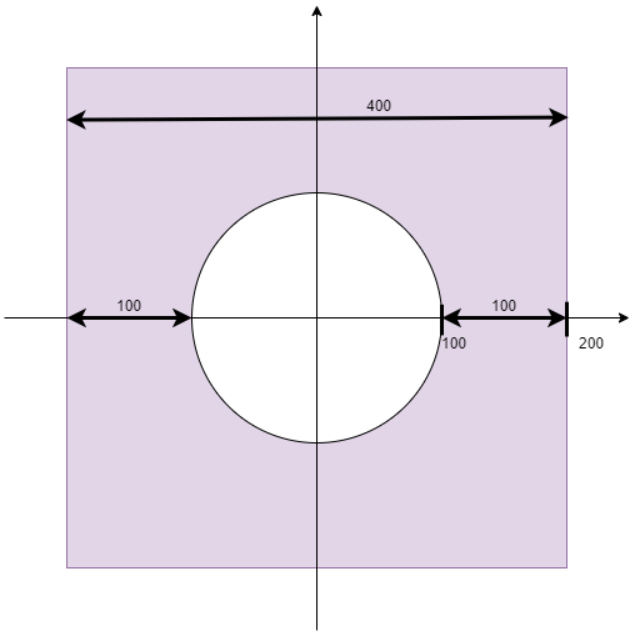

Рис 3.2.1 - фигура, представленная y-отрезками

Вместе с изменением представления фигуры меняется представление пространства листа. В данной реализации оно генерируется внутри сервера в предикате generate(), которому на вход подаются ширина и высота листа.

Листинг 3.2.8 - генерация пространства

```
generate(W,H,F):-
  findall(
    (I,Lst),
    (H1 #= H-0, W1 #= W-0,
    between(0,H1,I),
    Lst=[(0,W1)]), F).
```

Также был изменен предикат поиска ответа place_figures, на вход он принимает:

- данные о фигуре -
  - диапазоны dx, dy, angle поиска фигуры,
  - описанное выше представление фигуры;
- свободное пространство;
- размещение фигур;
- оставшееся свободное пространство.

Листинг 3.2.9 - располагает фигуры в заданном диапазоне

```
place_figures([],A,[],A):-true.
place_figures(_,[],_,_):-fail.
place_figures([(_,_,_,[] )|_], _, _, _):-fail.

place_figures([(Dx,Dy,(AngleMin,AngleMax),[(Angle,Delta,(MinY
  ,MaxY),H)|_] )|L], F,
    [json{xCenter:X, yCenter:Y, angle: Angle}|Ans], QRes):-
    between(AngleMin,AngleMax,Angle),
    fromField3_helper(Lst3, Delta,H, F,Dx,Dy,(MinY,MaxY)),
    member((Y,X),Lst3),
     delete0(F,H,(X,Y),F2),
     place_figures(L,F2,Ans,QRes).
place_figures([(Dx,Dy,Da,[_|Hs])|L],F,Ans,QRes):- place_figu
  res([(Dx,Dy,Da,Hs)|L],F,Ans,QRes).
```

3.2.4 Масштабирование

В данной реализации используется описанный выше способ представления фигур y-отрезками. Однако вместо поиска ответа в исходном размере, происходит итеративное увеличение размера с учетом ранее найденных расположения фигур.

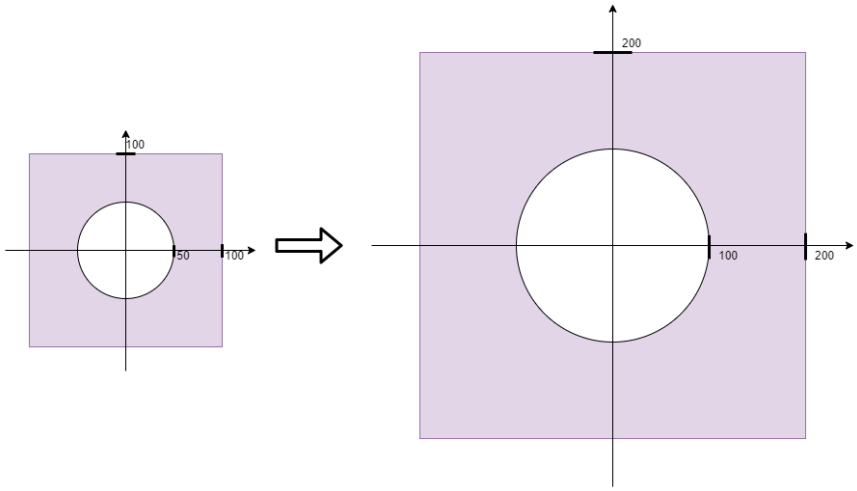

Рис 3.2.2 - итеративный процесс масштабирования

Листинг 3.2.10 - пример сгенерированного запроса для итеративного увеличения масштаба

```
1. myQuery(Ans1):-
2.    Width0 is 375,Width1 is 750,
3.    Height0 is 125,Height1 is 250,
4.    Scale0 is 0.5,Scale1 is 1,
5.    generate(Width0, Height0, Field0),
6.    fig0(Fig00,Scale0),
7.    place_figure_with_angles([((0,Width0),(0,Height0), (0,359),F
   ig00),((0,Width0),(0,Height0), (0,359),Fig00),((0,Width0),(0,He
   ight0), (0,359),Fig00)],Field0, Ans0, _),
8.    generate(Width1, Height1, Field1),
9.    fig0(Fig01, Scale1),
10.          Kscale1 is Scale1/Scale0,
11.          get_next_approc_coords(Kscale1,[Fig01,Fig01,Fig01], A
   ns0, ApprocFigNewScale1),
12.          place_figure_with_angles(ApprocFigNewScale1,Field1,An
   s1,_).
```

На листинге 3.2.10 показан запрос для поиска ответа в двух масштабах. Сначала идет определение размеров масштабов (строки 2-4). После - итеративный поиск. Для первого размера (строка 7) происходит вызов предиката place_figure_with_angles, который находит решение (Ans0), которое в предикате get_next_approc_coords преобразуется в аппроксимированное положение фигур для следующего размера, в строке 12 происходит поиск следующего решения.

Листинг 3.2.11- предикат преобразования координат фигуры

```
1.  % Преобразует старые координаты в новые (для следующего масшта
    ба)
2.  get_fig_approc_coords(Offset, Kscale, Fig, FigPos, ScaledAns)
    :-
3.          MinX is floor(FigPos.xCenter * Kscale) - Offset,
4.          MaxX is ceiling(FigPos.xCenter * Kscale) + Offset,
5.          MinY is floor(FigPos.yCenter * Kscale) - Offset,
6.          MaxY is ceiling(FigPos.yCenter * Kscale) + Offset,
7.          MinAng is FigPos.angle - 3,
8.          MaxAng is FigPos.angle + 3,
9.          ScaledAns = ((MinX, MaxX), (MinY, MaxY), (MinAng, MaxA
    ng), Fig).
```

Листинг 3.2.12 - предикат преобразования ответа меньшего масштаба в аппроксимированный ответ для следующего

```
1. get_next_approc_coords(_, [], [], []).
2. get_next_approc_coords(Kscale,[CurFig|TailFigs],[CurFigAns|Ta
   ilAns],[CurApproc|TailApproc]) :-
3.          Offset is 30,
4.          get_fig_approc_coords(Offset, Kscale, CurFig, CurFigA
   ns, CurApproc),
5.          get_next_approc_coords(Kscale, TailFigs, TailAns, Tai
   lApproc).
```

3.3 Алгоритм размещения фигур

Ниже показан листинг функции, которая принимает решение о размещении фигуры на лист. На вход ей подается:

- список индексов фигур (фигуры уже загружены на сервер под нужными индексами) - data;
- список коэффициентов масштабирования - scaleCoefs;
- списки размеров листа соответствующих масштабов - w, h;
- аргумент result является рекурсивно обновляемым списком листов с текущем найденным размещением.

При следующем рекурсивном вызове функции GetWorkingArrangementPreDefFigs (строки 32, 41) аргумент data уменьшается на одну фигуру, а в result наоборот, добавляется фигура в один из листов. Если не было найдено размещения ни в одном из листов, фигура добавляется в новый.

Листинг 3.3.1 - функция поиска решения

```
1. public static List<List<int>>
   GetWorkingArrangementPreDefFigs(List<int> data, List<double>
   scaleCoefs, List<int> w, List<int> h,
2.     List<List<int>> result = null)
3. {
4.     if (result == null)
5.     {
6.         result = new List<List<int>> { new List<int>() };
7.
8.         result[0].Add(data[0]); // первую фигуру всегда в
   новый лист
9.         data.RemoveAt(0);
10.     }
11.     if (data.Count == 0)
12.     {
13.         logger.Log(result);
14.         return result;
15.     }
16.
17.     int currentFig = data[0];
18.     var nextData = new List<int>(data);
19.     nextData.RemoveAt(0);
20.
21.
22.     // Пытаемся последовательно добавлять в уже существующие
   листы
23.     for (int i = 0; i < result.Count; i++)
24.     {
25.         var newCurLst = new List<int>(result[i]) {
   currentFig };
26.
27.         logger.Log(newCurLst);
28.
29.         if (DoesCurrentListFitPreDefFigs(newCurLst,
   scaleCoefs, w, h))
30.         {
31.             result[i].Add(currentFig);
32.             var curResult =
   GetWorkingArrangementPreDefFigs(nextData, scaleCoefs, w, h,
   result);
33.             if (curResult != null)
34.                 return curResult;
35.         }
36.     }
37.
38.     // Если не получилось добавить к существующим, кладем в
   новый
39.     var tmp = new List<int> { currentFig };
40.     var newResult = new List<List<int>>(result) { tmp };
```

```
41.     var curRes = GetWorkingArrangementPreDefFigs(nextData,
  scaleCoefs, w, h, newResult);
42.
43.     return curRes ?? null;
44. }
```

3.4 Использование Microsoft Azure

Т.к. задача - ускорить вычисления за счет параллельного подсчета. Было решено использовать облачную платформу Microsoft Azure. Для студентов доступны 100$ кредита на пользование платформой. Для целей диплома эти средства потрачены на вычислительные мощности.

Задача подразумевает развертывание неопределенного числа серверов, каждый из которых требует установку зависимостей (swi prolog), перенос исходных файлов и непосредственно запуск сервера. Для упрощения данного процесса был использован Docker и проделаны следующие шаги:

1. модифицирован код сервера для создания образа: добавлены новые библиотеки и изменена структура вызова инициализирующей функции;
2. создан docker файл с инструкциями по запуску сервера;
3. создан образ сервера;
4. опубликован на dockerHub как публичный репозиторий.

Листинг 3.3.1 - содержимое файла конфигурации образа

```
1.  FROM swipl
2.
3.  COPY . /app
4.  EXPOSE 8888
5.
6.  ENTRYPOINT ["swipl"]
7.  CMD ["/app/server.pl","--user=daemon", "--fork=false", "--
    port=8888"]
```

Из листинга 3.3.1 видно, что образ создается из базового swipl - образа, все содержимое корневой, для файла конфигурации, папки (код сервера, вспомогательные файлы) переносится в папку app в контейнере. При запуске контейнера параметры передаются команде, указанной в entrypoint - запускается

swipl. Инструкция cmd указывает, какую команду необходимо запустить, когда контейнер запущен - во время запуска контейнера будет запущен сервер-демон на порту 8888. [26]

Для совместимости с docker инициализацию сервера пришлось изменить. На листинге 3.3.2 инициализация происходила при помощи initialization, а значение порта было захардкожено. На листинге 3.3.3 файл сервера является модулем, при инициализации которого можно указать порт.

Листинг 3.3.2 - изначальная версия инициализации сервера

```
1. :- initialization server.
2. server() :-
3.     http_server(http_dispatch, [port(8080)]).
```

Листинг 3.3.3 - измененная версия инициализации сервера

```
1. :- module(server,
2.       [ server/1            % ?Port
3.       ]).
4. server(Port) :-
5.  http_server(http_dispatch, [port(Port)]).
```

Листинг 3.3.4 - команды создания и запуска образа

```
1. docker build -t find-best-layout-server .
2. docker run -p 8888:8888 -d find-best-layout-server
```

На листинге 3.3.4 команда build отвечает за создание, а команда run за его запуск. При этом флаг -t задает название образа, -p связывает порты контейнера с портами хоста, -d терминал не прикрепляется к контейнеру (контейнер работает в фоновом режиме).

3.5 Пример работы

Пусть на вход дано пять типов фигур (рис 3.4.1): один большой, три средних, один маленький. Каждое из изображений размера 8645x2976.

Рис 3.5.1 - входные изображения

Для соответствующего набора изображений конфигурация настроена таким образом, что:

- размер пространства (ширина и высота) - 7500x2500;
- цвет, считываемый с изображения фигур - #9B9B9B (серый);
- количество деталей - для данного примера 6 деталей каждого типа;
- шаг поворота фигур - 3 градуса;
- коэффициенты масштабирования - 0.5, 1;
- отступ от границы фигуры - 0 (нет отступа);
- таймаут сервера - 5 минут.

На рис. 3.4.2 можно наблюдать за процессом поиска решения. Изначально происходит загрузка фигур, предобработка заканчивается спустя 8 секунд. Далее идет проверка первого размещения - две большие фигуры, решение найдено, попытка добавить следующую большую фигуру. Решение найдено не было, и третья большая фигура отправлена на второй лист. «Плохое» размещение запоминается и при повторной попытке проверить такое же, сразу считается недопустимым. По итогу программа выводит найденное размещение. Для этого примера потребовалось три листа, большая часть из фигур разместилась на первом.

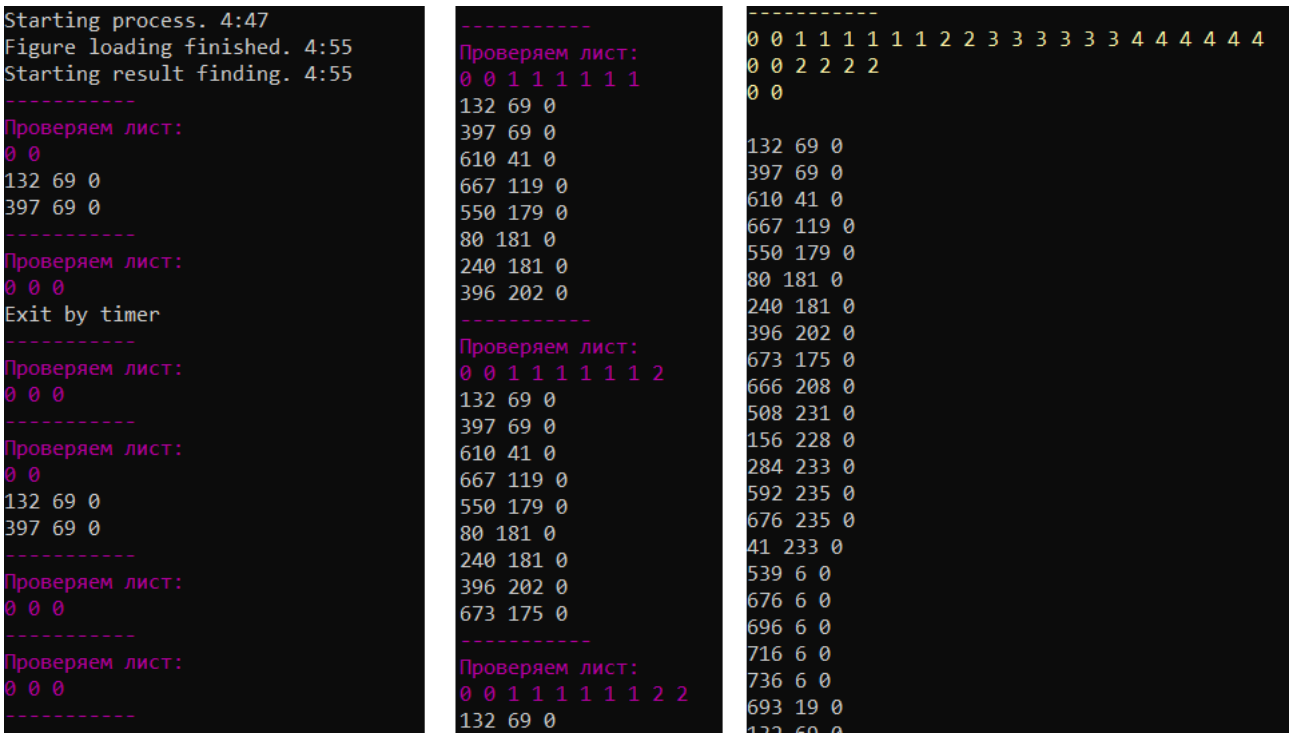

Рис 3.5.2 - пример вывода программы

На рис 3.4.3 и 3.4.4 показаны результаты размещения в графической и текстовой формах.

Рис 3.5.3 - вывод результата в графической форме

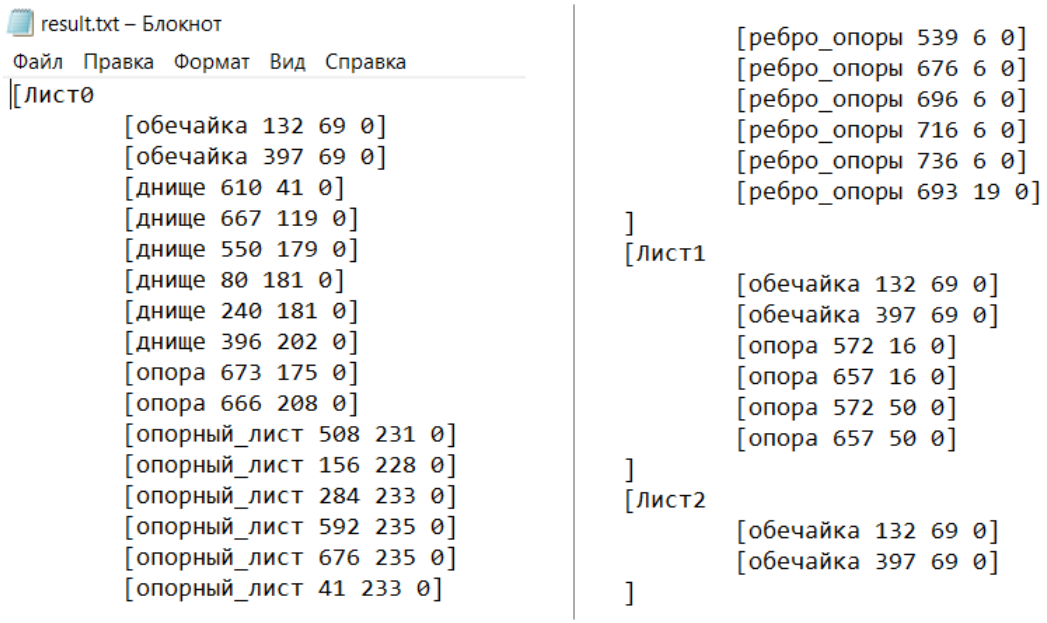

Рис 3.5.4 - сформатированный вывод размещения

3.6 Пример работы для параллельного случая

Для проверки реализации параллельного случая в качестве тестовых данных были использованы те же детали с рис. 3.5.1, а также похожая конфигурация: отличалась она количеством серверов (в данном случае два) и добавлением таймаутов для запуска побочной ветки решений.

Данная конфигурация интересна для параллельного решения тем, что дважды происходит выход по таймеру, а в конце время поиска решений увеличивается (ввиду большого числа фигур на листе).

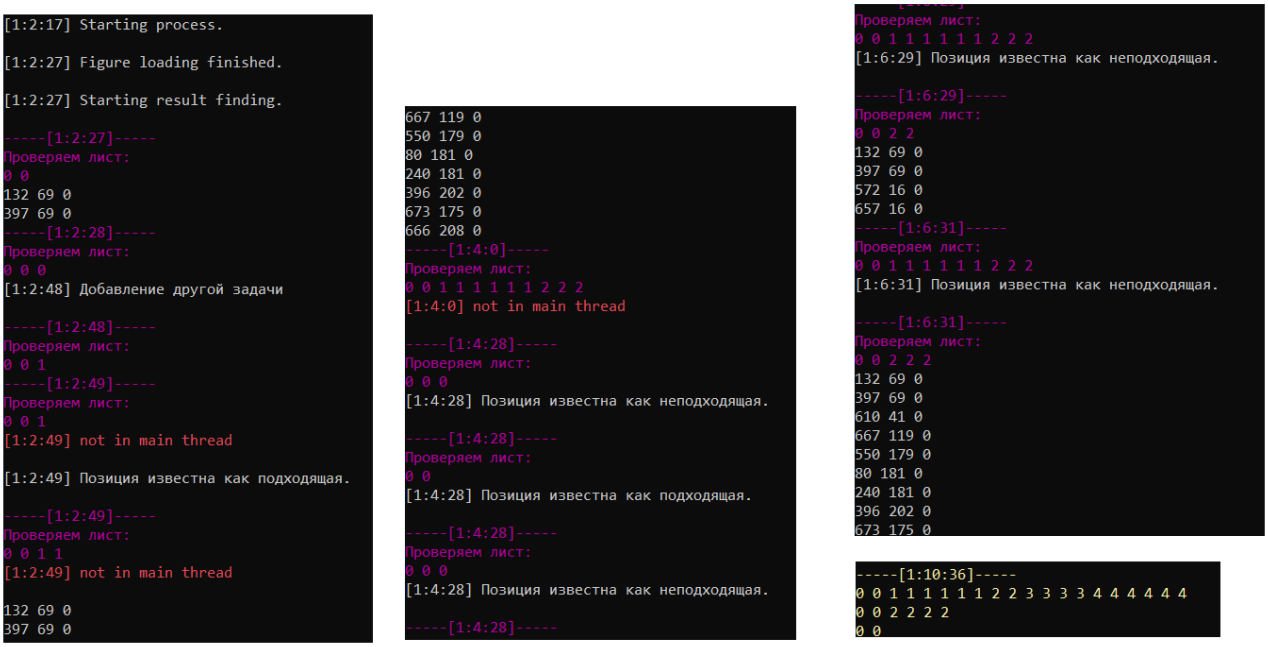

Рис 3.6.1 - пример вывода параллельной программы

На рис. 3.6.1 продемонстрированы ключевые моменты параллельного поиска решения:

- предобработка фигур;
- проверка комбинаций, пока приемлемое время;
- в 01:02:28 был отправлен запрос на 3 большие фигуры, за время таймаута он не был обработан и была запущена дополнительная ветка решений;
- 2 минуты работала дополнительная ветка, успев разместить 12 деталей на первом листе, к этому моменту основная ветка решений завершилась с ответом «не подходящее»;
- основная ветка решений продолжила свою работу, фактически с того места, где закончила работу дополнительная: нужные размещения уже были обработаны, а результат сохранен;
- в конце, спустя 8 минут от запуска, было найдено решение аналогичное найденному последовательной реализацией.

3.7 Выводы из технологического раздела

В данном разделе были выбраны язык программирования и среда разработки, продемонстрированы собственные реализации и описано взаимодействие с Microsoft Azure.

В качестве языка программирования был выбран C#, в качестве среды разработки была выбрана «Visual Studio 2019», серверная часть была реализована на swi-prolog т.к. это логический язык программирования, решающий задачу полного перебора. Для хостинга серверов выбрана облачная платформа Microsoft Azure. Для ускорения процесса настройки сервера, использован Docker.

Описана пиксельная реализация решения задачи и ее оптимизация «опорные точки», а также реализация через представление сегментами и последующее ускорение при помощи итеративного масштабирования. Продемонстрированы примеры работы программы: последовательная и параллельная реализации.

4 Исследовательский раздел

В данном разделе будет проведено сравнение реализаций поиска и его оптимизаций.

При тестировании программы на фигурах размера порядка десятка тысяч пикселей в ширину сервер выдавал ошибку памяти, причем было установлено максимальное значение. Ввиду этого принято решение ограничить размер входных фигур до 1000x1000 пикселей. При необходимости входные изображения большего масштаба можно отмасштабировать.

4.1 Идеальный случай

- большой прямоугольник - 400x85 пикселей
- звезда, помещается в отверстие прямоугольника с запасом.

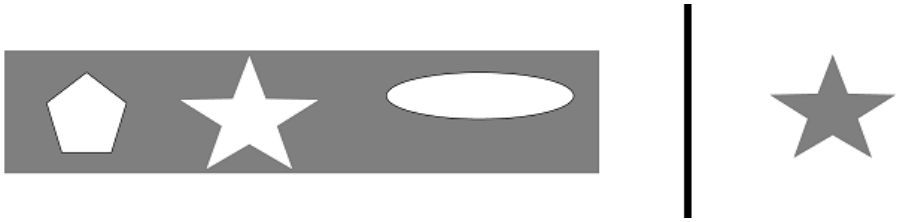

Рис 4.1.1 - детали для идеального случая

4.1.1 Пиксельная реализация

Было замерено время поиска **всех** решений на листе 402x88:

- фигуры без поворотов - 40 секунд;
- фигуры по 3 поворота - 1 минута (найдено 660 решений);
- фигуры по 90 поворотов - 1 час (6696 решений).

В данном эксперименте наибольшая фигура практически полностью занимает пространство листа, что заметно сужает время перебора для второй фигуры

4.1.2 Оптимизация опорные точки

В данной оптимизации, на этапе предобработки были найдены 4 опорные точки для каждой из фигур. Таким образом были упрощены правила подбора размещения. Если опорные точки не помещаются, то текущее расположение считается недействительным.

На конфигурации идеального случая, с листом размера 402x88 пикселей были получены следующие результаты:

- 3 поворота: 1 минута -> 17 секунд
- 90 поворотов: 1 час -> 13 минут

За счет данной оптимизации скорость поиска всех размещений уменьшилась в 3 раза.

Размещение на конфигурации идеального случая показала, что даже при таком маленьком размере листа перебор всех решений занимает огромное время. Принято решение в дальнейшем тестировать поиск до первого найденного решения.

4.2 Хороший случай

На рис. 4.2.1 показаны детали, использующиеся в конфигурации хорошего случая. Исходные изображения 8645x2976 пикселей, лист: 7500x2500 пикселей. Уменьшены в 10 раз. Повороты фигур в 3 градуса. Состав поиска: 2 обечайки, 3 днища, 30 ребер опоры.

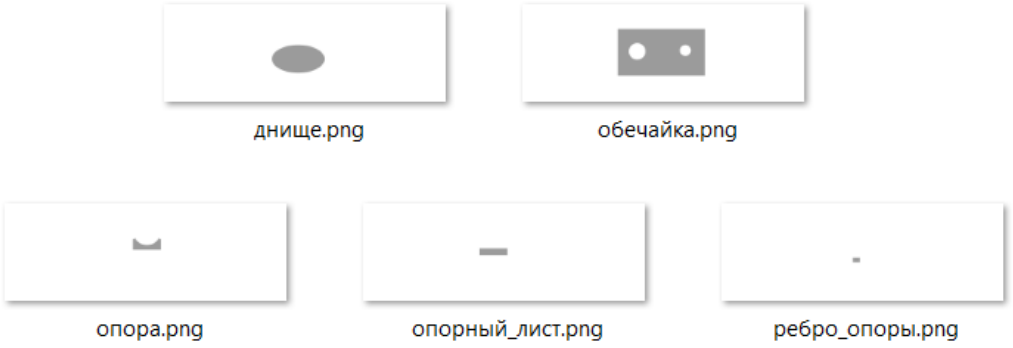

Рис 4.2.1 - набор деталей

При попытке запуска программы на хорошей конфигурации решение не было найдено за 20 минут, выполнение пришлось прервать. В ходе анализа различных результатов групп фигур из этой конфигурации выяснилось, что при предварительной сортировке фигур по размеру можно достичь значительного ускорения. После предварительной сортировки деталей, решение было найдено за 12 секунд.

Также было произведено сравнение двух реализаций пролога: SICStus prolog и SWI-Prolog на хорошей конфигурации.

Результаты:

- SICStus prolog - 25 секунд;
- SWI-Prolog - 12 секунд.

Смены языка произведено не было ввиду того, что SWI-Prolog для этой задачи работает в 2 раза быстрее.

В параллельной реализации эта комбинация фигур не имела преимуществ: каждая комбинация проверялась менее чем за секунду, дополнительный сервер использован не был.

4.3 Приемлемый случай

Конфигурация данного случая представляет из себя 6 деталей каждого типа из рис. 4.2.1.

Был развернут 1 инстанс созданного контейнера сервера на портале Microsoft Azure. Адрес машины была прописан в конфигурационный файл. На рис. 4.4.2 видно, что в 23:20 было получено 25 килобайт данных — это произошла загрузка данных о деталях. Далее начался процесс поиска размещения. Всего было отправлено 25 запросов. Поиск занял 13 минут.

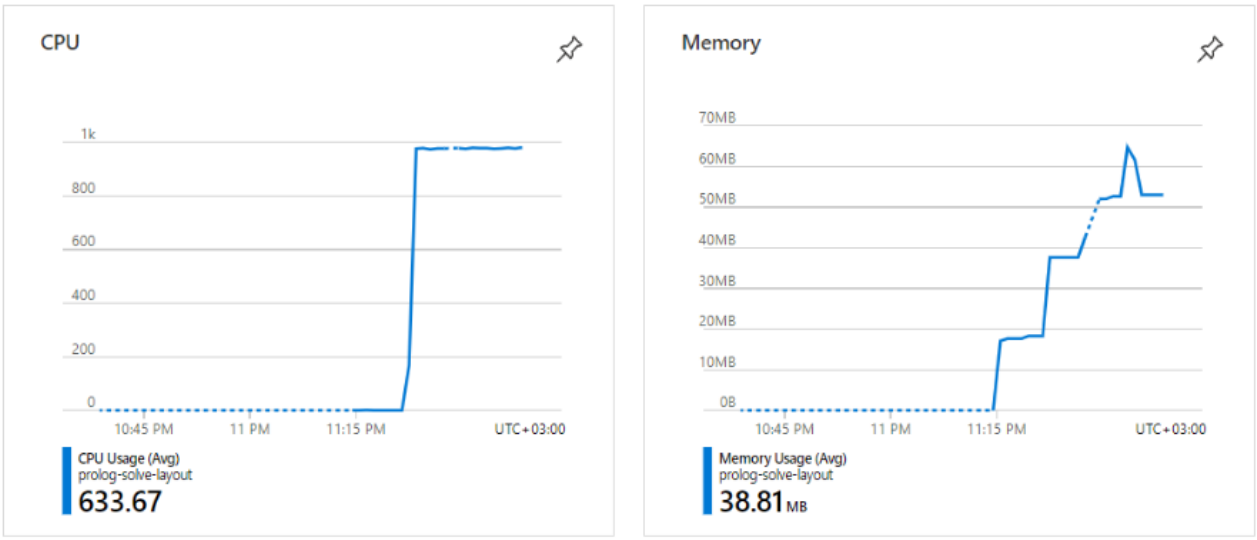

Рис 4.3.1 - показатели использования времени ЦП и памяти

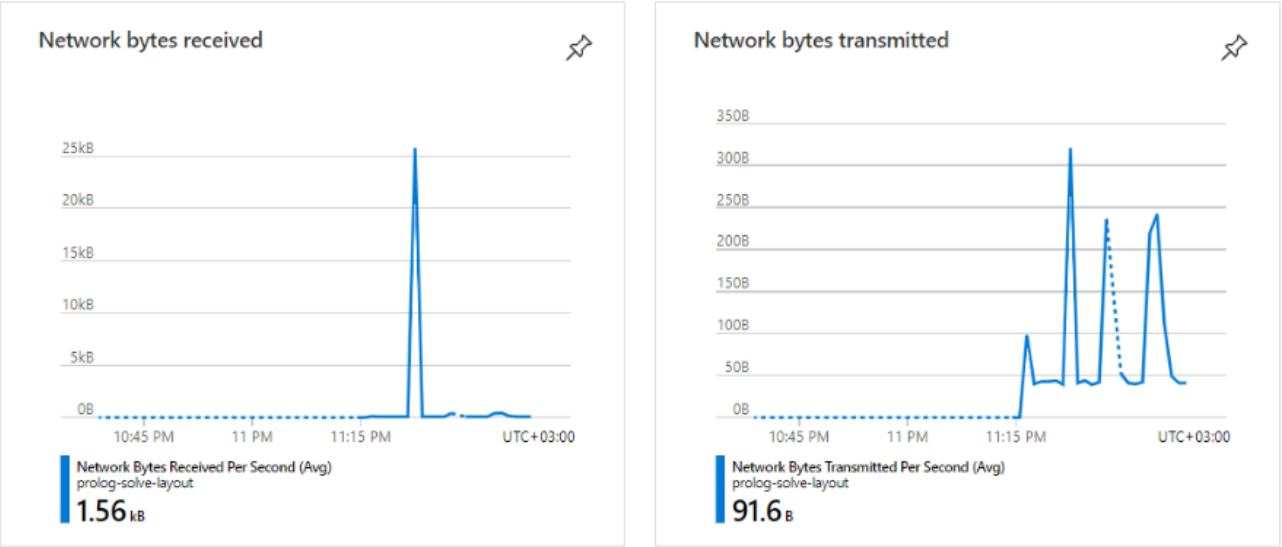

Рис 4.3.2 - количество полученных(слева) и переданных(справа) байтов

Для приемлемого случая также было проведено сравнение скорости реализации пролога: SWI-Prolog - 13 минут, SICStus - 16.

4.3.1 Параллельная реализация

Для исследования параллельной реализации были развернуты 2 инстанса созданного контейнера сервера на портале Microsoft Azure, их адреса внесены в файл конфигурации.

При выполнении поиска размещения для этого случая основную работу по поиску размещений выполнил дополнительный сервер. Это произошло ввиду того, что уже на второй комбинации основному серверу попалась неподходящая комбинация, первичное время ожидания было превышено и следующие 11 запросов выполнял дополнительный сервер, к этому моменту вычисления на основном сервере были закончены и управление конфигурациями передано обратно. Поиск занял 8 минут, что на 60% быстрее, чем последовательная реализация.

4.4 Нет решений

Для этого случая используется 3 детали «обечайка», показанные на рис. 4.2.1 размере листа 7500x2500. Фигуры также уменьшены в 10 раз.

Этот случай примечателен тем, что не имеется его размещения на одном листе. Полный перебор для доказательства этого факта занимает много времени (более 5 часов), принято решение прекращать поиск по определенному таймауту.

При использовании реализации представления «y-отрезки» данный случай решен не был. Полный перебор все еще недостаточно быстрый.

После внедрения итеративного увеличения масштаба была найдена проблема данной реализации: на листе 90x30 (масштаб 0.12) за 5 минут было найдено решение, которого в более крупных масштабах нет. На рис. 4.3.1 продемонстрировано, что при попытке разместить детали на свои места исходя из мелкого масштаба происходит наложение. Такая ситуация возникает в результате необходимости аппроксимации граничных точек при масштабировании. В случае если граничные точки исключить из мелких масштабов может случится описанная выше ситуация, в случае если их включить в решение, то фигура может, наоборот, оказаться слишком громоздкой.

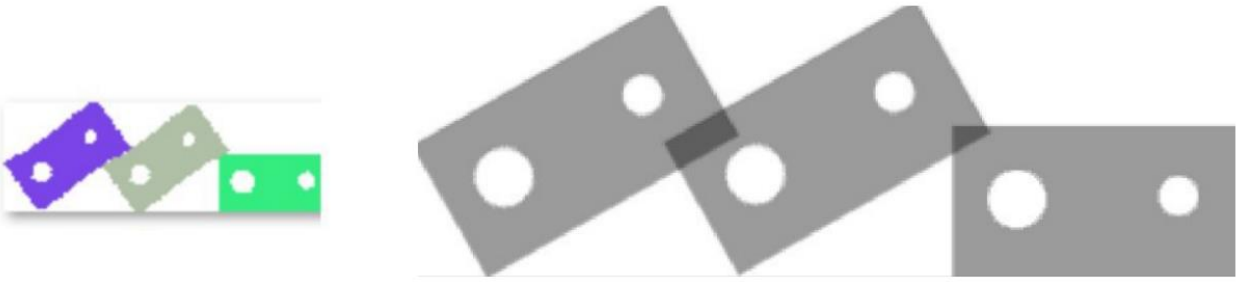

Рис 4.4.1 - найденное решение в разных масштабах

В параллельной реализации эта комбинация фигур не имеет никаких преимуществ: первое размещение (две фигуры) достаточно быстро найдено основным сервером, второе же (3 фигуры) не найдено за время таймаута, но дополнительному серверу невозможно отправить следующую комбинацию т.к. она является последней для проверки. Параллельная реализация будет давать преимущество только если этот случай - одна из комбинаций в поиске решения, как в приемлемом случае.

4.5 Визуализация результатов исследования

В данном разделе будут визуализированы результаты исследования.

На рис. 4.5.1 продемонстрировано различие между SWI-Prolog и SICStus реализациями пролога на разных конфигурациях. Видно, что для хорошего случая SWI-Prolog в 2 раза быстрее SICStus. Для приемлемого случая разница скоростей сокращается за счет конфигураций без решения, при исключении этих конфигураций, SWI-Prolog оказывается все также в 2 раза быстрее.

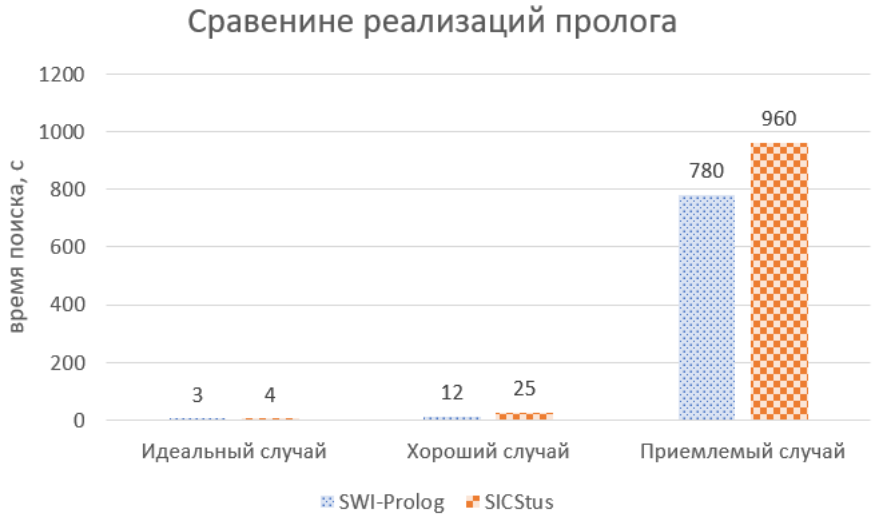

Рис 4.5.1 - сравнение скорости реализаций пролога

На рис. 4.5.2 было продемонстрировано различие по времени работы пиксельной реализации и реализации с ускорением опорными точками для поиска всех решений в зависимости от количества поворотов в фигурах.

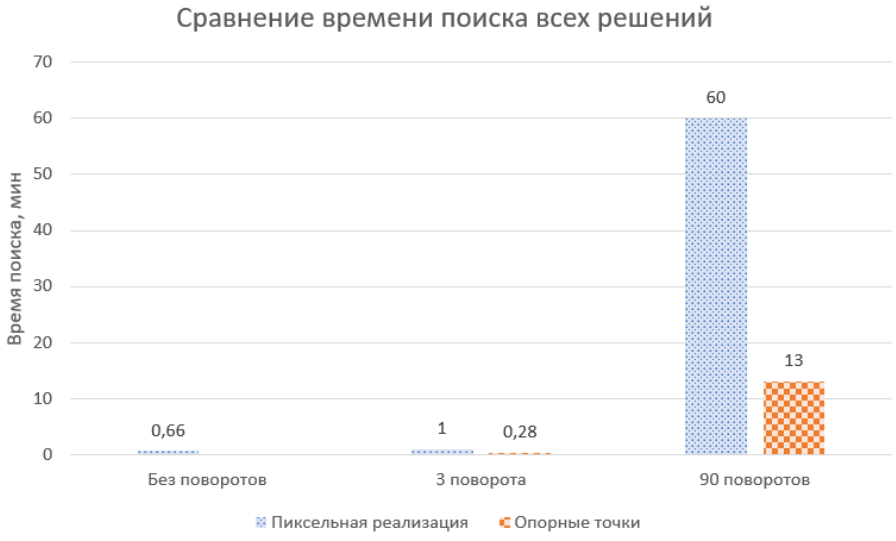

Рис 4.5.2 - сравнение скорости поиска всех решений

На рис. 4.5.3 продемонстрировано различие предложенных реализаций: для хорошего случая нет прироста в скорости, для приемлемого случая параллельная реализация на 60% быстрее обычной.

Рис 4.5.3 - сравнение предложенных реализаций

4.6 Выводы из исследовательского раздела

В данном разделе было проведено сравнение реализаций поиска и его оптимизаций на разных конфигурациях, визуализированы результаты. В ходе данного исследования было установлено, что:

- полный перебор всех комбинаций размещений слишком трудозатратная операция в любой из реализаций поиска;
- в данной задаче реализация SWI-prolog работает в 2 раза быстрее SICStus пролога;
- оптимизация «опорные точки» дает прирост в скорости в 3 раза;
- оптимизация «масштабирование» дает прирост в скорости в 3 раза;
- параллельная реализация не дает прироста скорости для идеального и хорошего случаев;
- для конфигурации без решений параллельная реализация дает прирост в скорости только в том случае, когда имеется более чем она компоновка внутри одной задачи поиска без решений;
- наилучшим же случаем для использования параллельной реализации является приемлемый набор, скорость поиска решения возрастает вплоть до 160%.

ЗАКЛЮЧЕНИЕ

Во время выполнения выпускной квалификационной работы была достигнута поставленная цель: разработан метод замощения поверхности, предложен способ его ускорения за счет параллельных вычислений.

В ходе выполнения поставленной цели, были решены следующие задачи:

- проведен анализ способов решения задач замощения;
- проведен анализ подходов к ускорению;
- разработан метод решения задачи о замощении;
- разработан метод распараллеливания алгоритма замощения;
- реализованы и протестированы разработанные методы.

В ходе выполнения поставленных задач были приобретены знания касательно способов задания плоских объектов, работы с облачным сервером Azure, настройкой Docker-контейнеров, выявлены достоинства и недостатки точных и приближенных алгоритмов раскроя.

Был предложен метод решения задачи о замощении, предполагающий использование SWI-Prolog сервера, для коммуникации и обработки изображений использовался C#.

Было предложено 3 метода распараллеливания и реализован один из них, предполагающий использование дополнительного сервера в случае, когда основной занят трудоемкими вычислениями.

Проведено исследование: были сравнены реализации пролога, предложенные методы решения, показана зависимость скорости поиска от количества поворотов в фигуре.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Минимизация отходов. [Электронный ресурс]. Режим доступа: https://ru.qaz.wiki/wiki/Waste_minimisation
2. Каюгина, С. М. Решение задач оптимального раскроя средствами MS Excel. // Молодой ученый. — 2016. — № 23 (127). — С. 54-57.
3. Петренко, С. В. Оптимизация размещения двумерных геометрических объектов на анизотропном материале с использованием методов математического программирования.
4. Silvano Martelo, Paolo Toth. Knapsack problems. — Great Britain: Wiley, 1990. — 306 с. — ISBN 0-471-92420-2.
5. А. В. Смирнов. О задаче упаковки в контейнеры. УМН, 1991, том 46, выпуск 4(280), страницы 173–174.
6. Лежнев А.В. Динамическое программирование в экономических задачах. М.: 2010 с. 58-60
7. Туровец О.Г., Родионов В.Б., Бухалков М.И. Организация производства и управление предприятием. ИД «ИНФРА-М», 2007. Глава 10.2
8. Официальный сайт программы Redcafe. Редактор или САПР выкроек одежды. [Электронный ресурс]. Режим доступа: http://redcafestore.com/
9. T-FLEX PLM | Конструкторская подготовка производства и анализ изделий (CAD/CAE). Работа с различными CAD-системами. [Электронный ресурс]. Режим доступа: https://www.tflex.ru/products/docs/page5/
10. САПР. Типовые методы геометрического моделирования объектов проектирования. Государственный комитет СССР по стандартам, 1988.
11. Аввакумов В.Д. Метод стыковки плоских фигур сложной геометрической формы.
12. Канторович Л.В. 1971 Рациональный раскрой промышленных материалов. Глава 3
13. Головистиков А.В. задачи двумерной прямоугольной упаковки и раскроя
14. Наумович, Н.А. Экспериментальное исследование серий градиентных методов // Методы, алгоритмы и программы решения экстремальных

задач: сб. науч. тр. – Минск: Ин-т техн. кибернетики АН БССР, 1985. – С. 98–111.

15. Н.Н. Попова. Суперкомпьютерное моделирование и технологии. Лекция 5. Распределение данных. Основы MPI, 21.09.2016.
16. Оленев Н.Н. Основы параллельного программирования в системе MPI. М.: ВЦ РАН. 2005. 80 с.
17. Грид-система, национальная библиотека им. Н.Э.Баумана. [Электронный ресурс]. - Режим доступа: https://ru.bmstu.wiki/Грид-система
18. Foster What is the Grid? A Three Point Checklist // GridToday. — 2002. — Июль
19. Frederic Magoules, Jie Pan, Kiat-An Tan, Abhinit Introduction to Grid Computing. – CRC Press, 2009. – С. 4-8.
20. К.В. Семёнов, А.Н. Антропов, Грид-системы и потенциал их использования. Вестник Инновационного Евразийского университета, 2014.
21. А.П. Демичев, В.А. Ильин, А.П. Крюков 2007, Введение в грид-технологии с. 9
22. Вычисления на GPU, Nvidia. [Электронный ресурс].- Режим доступа: https://www.webcitation.org/684cRJCyL?url=http://www.nvidia.ru/page/gpu_computing.html
23. Буряк Д.Ю, вычислительные платформы для реализации НС, Лекция 10, 2021. [Электронный ресурс]. – Режим доступа: http://sqi.cs.msu.ru/files/nnbase/2020_2021/lecture10.pdf
24. C# документация. Microsoft Docs. [Электронный ресурс]. – Режим доступа: https://docs.microsoft.com/en-us/dotnet/csharp/
25. Swi-prolog документация. [Электронный ресурс]. – Режим доступа: https://www.swi-prolog.org/pldoc/man?section=httpserver
26. Docker документация. [Электронный ресурс]. – Режим доступа: https://docs.docker.com/engine/reference/builder/

ПРИЛОЖЕНИЕ А

Размещение фигур обычная реализация

Листинг А.1 - класс последовательной реализации поиска размещения

```
1. using System.Collections.Generic;
2. using DataClassLibrary;
3.
4. namespace SolveTask
5. {
6.   public class SolutionChecker : SolutionCheckerBase
7.   {
8.     /// <summary>
9.     /// Расположение фигур на одном листе
10.     /// </summary>
11.     private ResultData FitCurrentListArrangement(List<int>
   figInd, List<double> scaleCoefs, List<int> w, List<int> h)
12.     {
13.       if (positions.IsPosBad(figInd))
14.       {
15.         logger.Log("Позиция известна как неподходящая.");
16.         return null;
17.       }
18.
19.       if (positions.IsPosGood(figInd))
20.       {
21.         logger.Log("Позиция известна как подходящая.");
22.         return positions.GetGoodPosition(figInd);
23.       }
24.
25.       ResultData res = prologCluster.GetAnyResult(w, h,
   scaleCoefs, figInd);
26.       if (res == null)
27.       {
28.         positions.AddBadPos(figInd);
29.         return null;
30.       }
31.
32.       positions.AddGoodPos(figInd, res);
33.       logger.Log(res);
34.
35.       return res;
36.     }
37.
38.     /// <summary>
39.     /// Рекурсивное расположение всех фигур
40.     /// </summary>
41.     public override List<List<int>>
   GetWorkingArrangement(List<int> data, List<double>
   scaleCoefs, List<int> w, List<int> h,
42.       List<List<int>> result = null)
```

```
43.     {
44.       if (result == null)
45.       {
46.         result = new List<List<int>> { new List<int>() };
47.
48.         result[0].Add(data[0]); // первую фигуру всегда в
  новый лист
49.         data.RemoveAt(0);
50.       }
51.       if (data.Count == 0)
52.       {
53.         logger.Log(result);
54.         return result;
55.       }
56.
57.       int currentFig = data[0];
58.       var nextData = new List<int>(data);
59.       nextData.RemoveAt(0);
60.
61.
62.       // Пытаемся последовательно добавлять в уже
  существующие листы
63.       for (int i = 0; i < result.Count; i++)
64.       {
65.         var newCurLst = new List<int>(result[i]) {
  currentFig };
66.
67.         logger.Log(newCurLst);
68.
69.         if (FitCurrentListArrangement(newCurLst, scaleCoefs,
  w, h) != null)
70.         {
71.           result[i].Add(currentFig);
72.           var curResult = GetWorkingArrangement(nextData,
  scaleCoefs, w, h, result);
73.           if (curResult != null)
74.             return curResult;
75.         }
76.       }
77.
78.       // Если не получилось добавить к существующим, кладем
  в новый
79.       var tmp = new List<int> { currentFig };
80.       var newResult = new List<List<int>>(result) { tmp };
81.       var curRes = GetWorkingArrangement(nextData,
  scaleCoefs, w, h, newResult);
82.
83.       return curRes ?? null;
84.     }
85.
86.     /// <summary>
87.     /// Известно разделение фигур по листам, ищется
  расположение
```

```
88.     /// </summary>
89.     public List<ResultData>
  PlacePreDefinedArrangement(List<List<int>> arrangement, int
  wLast, int hLast, List<double> scaleCoefs)
90.     {
91.       List<ResultData> results = new List<ResultData>();
92.       foreach (List<int> figInd in arrangement)
93.       {
94.         FillLists(wLast, hLast, out var w, out var h,
  scaleCoefs);
95.
96.         var res = FitCurrentListArrangement(figInd,
  scaleCoefs, w, h);
97.         if (res == null)
98.         {
99.           logger.LogError("Ошибка: невозможно расположиться
  фигуры при заданном распределении");
100.                return null;
101.              }
102.              else
103.              {
104.                results.Add(res);
105.              }
106.            }
107.            return results;
108.          }
109.        }
110.      }
111.
```

ПРИЛОЖЕНИЕ Б

Размещение фигур параллельная реализация

Листинг Б.1 - класс параллельной реализации поиска размещения

```
1. using System;
2. using System.Collections.Generic;
3. using DataClassLibrary;
4. using SolveTask.Server;
5. using System.Threading;
6. using System.Threading.Tasks;
7. using System.Net.Http;
8.
9. namespace SolveTask
10. {
11.   /// <summary>
12.   /// Если за определенное время не было найдено
  расположение начинаем подсчитывать следующие
13.   /// </summary>
14.   public class ParallelTimeoutSolutionChecker :
  SolutionCheckerBase
15.   {
16.     TimeSpan timeout = TimeSpan.FromSeconds(20);
17.     Dictionary<Task, (List<int>, List<int>, List<List<int>>,
  bool)> tasksToArrangement;
18.
19.     List<double> scaleCoefs;
20.     List<int> w;
21.     List<int> h;
22.
23.     /// <summary>
24.     /// Расположение фигур на одном листе
25.     /// </summary>
26.     private ResultData FitCurrentListArrangement(List<int>
  figInd, List<int> backedData = null, List<List<int>>
  backedResult = null)
27.     {
28.       if (!Program.IsMainThread)
29.         logger.LogError("not in main thread");
30.
31.       if (positions.IsPosBad(figInd))
32.       {
33.         logger.Log("Позиция известна как неподходящая.");
34.         return null;
35.       }
36.
37.       if (positions.IsPosGood(figInd))
38.       {
39.         logger.Log("Позиция известна как подходящая.");
```

```
40.         return positions.GetGoodPosition(figInd);
41.       }
42.
43.       if (IsArrangementInProgress(figInd))
44.       {
45.         logger.Log("Данное размещение уже проверяется.");
46.         if (Program.IsMainThread)
47.         {
48.           logger.LogError("Т.к. в главной ветке, ожидаем
  результата.");
49.           var task = GetKeyByArrangement(figInd);
50.           var tmp = tasksToArrangement[task];
51.           tmp.Item4 = false;
52.           tasksToArrangement[task] = tmp;
53.           SpinWait.SpinUntil(() => task.IsCompleted);
54.           logger.LogError("Задача закончилась.");
55.           Thread.Sleep(TimeSpan.FromSeconds(2));
56.         }
57.         else
58.         {
59.           logger.LogError("Прекращаем выполнение в этой
  ветке.");
60.           Thread.CurrentThread.Abort();
61.         }
62.         return null;
63.       }
64.
65.       var findingTask = prologCluster.StartAnyResultTask(w,
  h, scaleCoefs, figInd);
66.       var finishedAfterFirstDelay = SpinWait.SpinUntil(() =>
  findingTask.IsCompleted, timeout); // ожидание таймаута
67.
68.       if (!finishedAfterFirstDelay && Program.IsMainThread)
69.       {
70.         var nextFigInd = MyCopy(figInd);
71.         nextFigInd[nextFigInd.Count - 1]++; //след фигура
  //TODO:a если нет
72.         logger.Log("Добавление другой задачи");
73.         logger.Log(nextFigInd);
74.         var nextTask = prologCluster.StartAnyResultTask(w,
  h, scaleCoefs, nextFigInd);
75.         tasksToArrangement.Add(nextTask, (nextFigInd,
  RemoveAllFirst(backedData), backedResult, true)); //other
  back?????
76.         nextTask.ContinueWith(delegate {
  HandleResult(nextTask); });
77.       }
78.
79.       SpinWait.SpinUntil(() => findingTask.IsCompleted);
80.       if (findingTask.IsCanceled || findingTask.IsFaulted)
81.       {
82.         positions.AddBadPos(figInd);
83.         return null;
```

```
84.       }
85.
86.       string responce =
  findingTask.Result.Content.ReadAsStringAsync().Result;
87.       var res = PrologServer.HandleServerResponce(responce);
88.
89.       if (res == null)
90.       {
91.         positions.AddBadPos(figInd);
92.         return null;
93.       }
94.
95.       positions.AddGoodPos(figInd, res);
96.       logger.Log(res);
97.
98.       return res;
99.     }
100.
101.          private bool IsArrangementInProgress(List<int>
  arrangement)
102.          {
103.            foreach (KeyValuePair<Task, (List<int>,
  List<int>, List<List<int>>, bool)> kvp in tasksToArrangement)
104.            {
105.              if
  (PlacementsStorage.IsTwoArrangementsEqual(arrangement,
  kvp.Value.Item1))
106.                return true;
107.            }
108.
109.            return false;
110.          }
111.
112.          private Task GetKeyByArrangement(List<int>
  arrangement)
113.          {
114.            foreach (KeyValuePair<Task, (List<int>,
  List<int>, List<List<int>>, bool)> kvp in tasksToArrangement)
115.            {
116.              if
  (PlacementsStorage.IsTwoArrangementsEqual(arrangement,
  kvp.Value.Item1))
117.                return kvp.Key;
118.            }
119.
120.            return null;
121.          }
122.          private List<int> RemoveAllFirst(List<int> list)
123.          {
124.            int first = list[0];
125.
126.            return list.FindAll((int el) => el != first);
127.          }
```

```
128.
129.          private ResultData
  HandleResult(Task<HttpResponseMessage> task)
130.          {
131.            // TODO: delete from in progress
132.            var backed = tasksToArrangement[task];
133.            var figInd = backed.Item1;
134.
135.            if (task.IsCanceled || task.IsFaulted)
136.            {
137.              positions.AddBadPos(figInd);
138.              return null;
139.            }
140.
141.            string responce =
  task.Result.Content.ReadAsStringAsync().Result;
142.            var res =
  PrologServer.HandleServerResponce(responce);
143.
144.            if (res == null)
145.            {
146.              positions.AddBadPos(figInd);
147.              return null;
148.            }
149.
150.            positions.AddGoodPos(figInd, res);
151.
152.            if (backed.Item4)
153.              GetWorkingArrangement(backed.Item2,
  backed.Item3);
154.
155.            return res;
156.          }
157.
158.          /// <summary>
159.          /// Рекурсивное расположение всех фигур
160.          /// </summary>
161.          public override List<List<int>>
  GetWorkingArrangement(List<int> data, List<double>
  scaleCoefs, List<int> w, List<int> h,
162.            List<List<int>> result = null)
163.          {
164.            this.scaleCoefs = scaleCoefs;
165.            this.w = w;
166.            this.h = h;
167.
168.            tasksToArrangement = new Dictionary<Task,
  (List<int>, List<int>, List<List<int>>, bool)>();
169.
170.            return GetWorkingArrangement(data, result);
171.          }
172.
```

```
173.          public List<List<int>>
  GetWorkingArrangement(List<int> data, List<List<int>> result
  = null)
174.          {
175.            if (result == null)
176.            {
177.              result = new List<List<int>> { new List<int>()
  };
178.
179.              result[0].Add(data[0]); // первую фигуру
  всегда в новый лист
180.              data.RemoveAt(0);
181.            }
182.            if (data.Count == 0)
183.            {
184.              logger.Log(result);
185.              return result;
186.            }
187.
188.            int currentFig = data[0];
189.            var nextData = new List<int>(data);
190.            nextData.RemoveAt(0);
191.
192.
193.            // Пытаемся последовательно добавлять в уже
  существующие листы
194.            for (int i = 0; i < result.Count; i++)
195.            {
196.              var newCurLst = new List<int>(result[i]) {
  currentFig };
197.
198.              logger.Log(newCurLst);
199.
200.              if (FitCurrentListArrangement(newCurLst,
  MyCopy(data), MyCopy(result)) != null)
201.              {
202.                result[i].Add(currentFig);
203.                var curResult =
  GetWorkingArrangement(nextData, result);
204.                if (curResult != null)
205.                  return curResult;
206.              }
207.            }
208.
209.            // Если не получилось добавить к существующим,
  кладем в новый
210.            var tmp = new List<int> { currentFig };
211.            var newResult = new List<List<int>>(result) {
  tmp };
212.            var curRes = GetWorkingArrangement(nextData,
  newResult);
213.
214.            return curRes ?? null;
```

```
215.           }
216.
217.           /// <summary>
218.           /// Известно разделение фигур по листам, ищется
  расположение
219.           /// </summary>
220.           public List<ResultData>
  PlacePreDefinedArrangement(List<List<int>> arrangement, int
  wLast, int hLast, List<double> scaleCoefs)
221.           {
222.             if (!Program.IsMainThread)
223.             {
224.               return null;
225.             }
226.
227.             List<ResultData> results = new
  List<ResultData>();
228.             foreach (List<int> figInd in arrangement)
229.             {
230.               FillLists(wLast, hLast, out var w, out var h,
  scaleCoefs);
231.
232.               var res = FitCurrentListArrangement(figInd);
233.               if (res == null)
234.               {
235.                 logger.LogError("Ошибка: невозможно
  расположиться фигуры при заданном распределении");
236.                 return null;
237.               }
238.               else
239.               {
240.                 results.Add(res);
241.               }
242.             }
243.             return results;
244.           }
245.         }
246.       }
247.
```

ПРИЛОЖЕНИЕ В

Поиск размещения

Листинг В.1 - модуль поиска размещения

```
1. :- use_module(library(dif)).    % Sound inequality
2. :- use_module(library(clpfd)).  % Finite domain constraints
3. :- use_module(library(chr)).    % Constraint Handling Rules
4. :- use_module(library(when)).   % Coroutining
5. :- use_module(lambda).
6. :- use_module(library(apply)).
7. :- use_module(library(thread)).
8.
9. % Задает пространство
10. % Ширина
11. % Высота
12. % Пространство
13. generate(W,H,F):-findall((I,Lst),
14.                     (H1 #= H-0, W1 #= W-0,
15.                     between(0,H1,I), Lst=[(0,W1)]), F).
16.
17. % Размещение фигур с поворотами в заданных диапазонах
18. % Массив фигур с диапазонами
19. % Поле для размещения
20. % Ответ - расположения
21. % Оставшееся пространство
22. place_figures_in_range([],A,[],A):-
  true. % все фигуры разместили
23. place_figures_in_range(_,[],_,_):-
  fail. % пространство пустое
24. place_figures_in_range([(_,_,_,[] )|_], _, _, _):-
  fail. % повороты закончились
25. place_figures_in_range([(Dx,Dy,(AngleMin,AngleMax),[(Angle,D
  elta,(MinY,MaxY),H)|_] )|L],
26.                     F,
27.                     [json{xCenter:X, yCenter:Y, angle: Angl
  e}|Ans],
28.                     QRes):-
29.     between(AngleMin,AngleMax,Angle),
30.     match_pre_fit(Lst3, Delta,H, F,Dx,Dy,(MinY,MaxY)),
31.     member((Y,X),Lst3),
32.     delete_figure_from_lst(F,H,(X,Y),F2),
33.     place_figures_in_range(L,F2,Ans,QRes).
34. place_figures_in_range([(Dx,Dy,Da,[_|Hs])|L],F,Ans,QRes):- p
  lace_figures_in_range([(Dx,Dy,Da,Hs)|L],F,Ans,QRes).
35.
36. % Предварительная проверка размещения по габаритам и централ
  ьной дельте
37. % (MinY,MaxY) -- мин и макс Y-полоски у фигуры
38. match_pre_fit(Lst3,Delta,H, F,(Xmin,Xmax),(Ymin,Ymax),(MinY,
  MaxY)):-
```

```
39.         bagof((Yi,Xs0),
40.                 (between(Ymin,Ymax,Yi),member((Yi,Xs0),F),
41.                 Yi22 is Yi +MaxY, Yi23 is Yi+MinY,
42.                 % что максимальные габариты помещаются
43.                 member((Yi22,Xs022),F),member((Yi23,Xs023),F
  )
44.                 % что в самых удалённых строках есть нужные
  зоны
45.                 , not(Xs022=[]), not(Xs023=[])
46.                 )
47.                 ,Lst),
48.         % что помещается Delta
49.         concurrent_maplist(\Z^Z2^(Z=(Yi,Xs),
50.                             findall((Yi,Xi),
51.                             (between(Xmin,Xmax,Xi),inXs2(Xs,
  Xi,Delta,Xs))
52.                             ,Z2)
53.                             ),Lst,Lst2),
54.         flatten(Lst2,Lst3), not(Lst3=[]).
55.
56. % Удаление детали из свободного пространства
57. % Свободное пространство (из чего)
58. % Деталь (что удаляем)
59. % Центр размещения фигуры
60. % Оставшееся свободное пространство (что получается)
61. delete_figure_from_lst([],[],_, [] ):-!.
62. delete_figure_from_lst(L,[],_, L ):-!.
63. delete_figure_from_lst([],_,_, _ ):-!,fail.
64. delete_figure_from_lst([(Y0,Lst0)|F],[(Yi,Lst)|L],(X,Y), [(Y
  0,Lst3)|F2] ):-
65.         Y0 #= Yi + Y,
66.         maplist(\Z^Z2^(Z=(X_s,X_e),X_s2 is X_s + X, X_e2 is
  X_e + X, Z2=(X_s2,X_e2)), Lst, Lst2),
67.         instert_segments(Lst0,Lst2,Lst3),!,
68.         delete_figure_from_lst(F,L,(X,Y),F2).
69. delete_figure_from_lst([L0|F],L,(X,Y), [L0|F2] ):-
70.         delete_figure_from_lst(F,L,(X,Y),F2).
71.
72. inXs2([(Xs,Xe)|_],X,D,Test):-[First|_]=D,
73.                             dist((Xs,Xe),Dist1),
74.                             dist(First,Dist2),
75.                             Dist1>=Dist2,
76.     between(Xs,Xe,X),conv2(X,D,D2),
77.     can_instert_segments(Test,D2).
78. inXs2([_|L],X,D,T):-inXs2(L,X,D,T).
79.
80. % Длина отрезка (A, B)
81. dist((A,B),R):-R is B-A.
82.
83. conv2(_,[],[]):-!.
84. conv2(Delta,[(X,X1)|Lst1],[(X_,X1_)|LstAns]):-
85.         X_ is Delta+X,
86.         X1_ is Delta+X1,
```

```
87.         conv2(Delta,Lst1,LstAns).
88.
89. % В уже разбитую полоску вставить несколько полосок
90. % куда, кого, [результат вставки]
91. instert_segments(L,[],L):-!.
92. instert_segments(L,Otr,A):-
  can_instert_segments(L,Otr),!, instert_segments_helper(L,Otr,
  A).
93. instert_segments_helper(L,[H|Otr],A):-
  ins2(L,H,A2), instert_segments_helper(A2,Otr,A).
94.
95. % Можно ли вставить сегменты
96. can_instert_segments(_,[]):-!.
97. can_instert_segments(L,Otr):- findall(Oi,(member(Oi,Otr),ins
  2(L,Oi,_)),Lst),
98.                 length(Lst,LL),
99.                 length(Otr,LO),
100.                       LL=LO.
101.
102.      ins2([H|L],Otr,A):- instert_segments(H,Otr,H2),!, appe
  nd(H2, L, A).
103.      ins2([H|L],Otr,[H|A]):- ins2(L,Otr,A).
104.
105.      % Преобразует старые координаты в новые (для следующег
  о масштаба)
106.      get_fig_approc_coords(Offset, Kscale, Fig, FigPos, Sca
  ledAns) :-
107.          MinX is floor(FigPos.xCenter * Kscale) - Offset,
108.          MaxX is ceiling(FigPos.xCenter * Kscale) + Offset,
109.          MinY is floor(FigPos.yCenter * Kscale) - Offset,
110.          MaxY is ceiling(FigPos.yCenter * Kscale) + Offset,
111.          MinAng is FigPos.angle - 3,
112.          MaxAng is FigPos.angle + 3,
113.          ScaledAns = ((MinX, MaxX), (MinY, MaxY), (MinAng,
  MaxAng), Fig).
114.
115.      get_next_approc_coords(_, [], [], []).
116.      get_next_approc_coords(Kscale, [CurFig|TailFigs], [Cur
  FigAns|TailAns], [CurApproc|TailApproc]) :-
117.             Offset is 30,
118.             get_fig_approc_coords(Offset, Kscale, CurFig,
  CurFigAns, CurApproc),
119.             get_next_approc_coords(Kscale, TailFigs, TailA
  ns, TailApproc).
120.
```