МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГБОУ ВПО «СЫКТЫВКАРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ ПИТИРИМА СОРОКИНА»
ИНСТИТУТ ТОЧНЫХ НАУК И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ
КАФЕДРА ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ
ТЕХНОЛОГИЙ В ОБРАЗОВАНИИ

УТВЕРЖДАЮ
Зав. кафедрой прикладной математики,
к.ф.-м.н., доцент,
_____Ермоленко А. В.
«____»_____ 2017г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Тема: «Генетические алгоритмы. Итеративный способ решения задачи раскроя»

Научный руководитель
к.ф.-м.н., доцент:
_____А. А. Холопов
«____»_____ 2017г.

Исполнитель, студент
145 группы:
_____Мельников Вадим
«____»_____ 2017г.

Сыктывкар 2017

Содержание

1. Введение в задачу раскроя материала

Экономия материала представляет собой сложную и важную проблему, с которой часто приходится встречаться на различных производствах, при резке различных материалов на: листы металла, стекла или дерева, трубы, профильный прокат, изделия сложной формы. Для её решения необходимо максимизировать использование материала, из которого вырезаются заготовки, что по сути и является рациональным раскроем материала. Максимизация использования материалов позволяет достичь большой экономии денежных средств.

На самом деле, задача раскроя является NP-полной даже для прямоугольников. Для фигур неправильной формы геометрическая сложность увеличивает количество совершаемых вычислений, поэтому применяются различные эвристические методы решения задачи.

2. Основные характеристика задач раскроя

Прежде чем приступать к рассмотрению алгоритмов решения задачи раскроя, следует рассмотреть характеристики, влияющие на то, как будет выглядеть итоговый алгоритм решения. В статье [1] Harald Dyckhoff приводит достаточно полное описание характеристик задач раскроя.

2.1. Пространственные характеристики

Основная характеристика раскроя — количество измерений:

— раскрой в одномерном пространстве;

— раскрой в двумерном пространстве;

— раскрой в трёхмерном пространстве.

Например загрузка поддонов является задачей в двумерном пространстве. В отличие от задач в двух и более измерениях, задача в одномерном

пространстве имеет явное решение. Достаточно подробно данная задача описывается в книге [2] Канторовича-Залгаллера — «Рациональный раскрой промышленных материалов». Так же в данной книге можно найти методы решения задач двумерного раскроя для случая прямоугольнных заготовок.

2.2. Количественные характеристики

Другая важная характеристика — количественная. В задаче раскроя необходимо некоторым образом измерять количественные и качественные характеристики фигур. Например, площадь, длина и ширина фигур. Или количество уже расположенных фигур. Тут можно рассмотреть два варианта:

— дискретное измерение с помощью, например, натуральных и целых чисел;

— дробное измерение на основе вещественных чисел.

Первый вариант позволяет нам подсчитывать количество изделий, уже расположенных на материале, а с помощью второго можно измерять различные характеристики фигур, такие как площадь, длина и ширина.

2.3. Геометрические характеристики

Не малую роль играют в раскрое сами фигуры, которые необходимо расположить на плоскости. В пространстве фигуры однозначно определяются с помощью следующих свойств:

— формой;

— размером;

— ориентацией;

2.4. Характеристики по ограничениям на результат

По ограничениям на результат можно выделить четыре основные группы:

— минимальное расстояние между объектами;

— ориентация фигур относительно друг друга;

— ограничение на количество фигур;

— ограничение на количество совершаемых «резов».

Автор [1] выделяет ещё несколько групп по различным признакам, но данные являются основными.

3. Постановка задачи

Прежде чем переходить к алгоритмам, применяемым для решения задачи раскроя, следует рассмотреть постановку задачи. Задача в двумерном пространстве ставится по аналогии с задачей раскроя в одномерном пространстве [3], хотя однозначно с помощью математических соотношений описать её нельзя.

Первоначально имеем множество фигур F и $|F| = n$, плоскость с шириной w и высотой h. Необходимо отыскать такое упорядоченное множество F', что $F' \subset F$ и $|F'| \to \max$, при условии что все фигуры из F' будут без пересечений размещены на плоскости с заданнами параметрами.

4. Методы представления фигур для раскроя

Самым видимым атрибутом задач раскроя и тем, с чем сразу сталкиваются исследователи в данной области — геометрическое представление фигур. Решение о том, как представлять фигуры, оказывает решающее значение на дальнейшую разработку системы.

4.1. Методы перемещения фигур

Прежде чем перейти к рассмотрению методов представления фигур, следует осветить важный вопрос, обычно опускаемый в литературе связанной с раскроем: «Каким образом перемещать фигуры относительно друг друга?».

Предположим, что мы уже некоторым образом расположили первую фигуру. Она распологается всегда в левом нижнем углу, как на рис. 1. Для удобства, границы контейнера изображать не будем. Условимся, что вторая фигура в конечном расположении будет закрашена серым цветом.

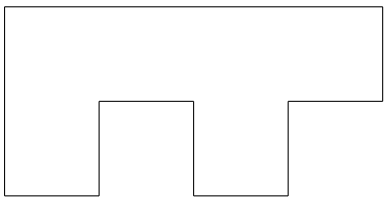

Рис. 1. Расположение первой фигуры

Самый первый метод, который будет интуитивно понятен всем — «лестничный» или же «тетрисный». Его суть заключается в том, что мы двигаем фигуру вниз до первого столкновения с другой, потом также влево, потом опять вниз, и так далее пока фигура не перестанет смещаться. Рассмотрим на примере, как «тетрисный» способ расположит следующую фигуру. На рис. 2 видно, что фигура перемещается слева направо с некоторым шагом по оси x и в итоге находит углебление внутри другой фигуры.

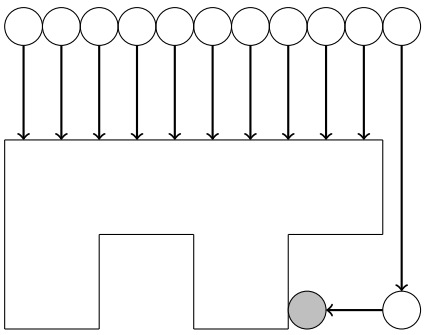

Рис. 2. Расположение второй фигуры «тетрисным» способом

Более сложный метод — «метод сквозного прохода». Его основная идея заключается в том, что фигуру просто перемещают сквозь остальные и ищут подходящее ей место. Как видно на рис. 3, данный метод нашёл закрытую полость, до которой предыдущий способ дойти не смог.

Самый сложный метод — движение вдоль контура. Под контуром, в данном случае, подразумевается обновляемый контейнер. Изначально имеется некоторый контур пустого контейнера, затем в него добавляются одна

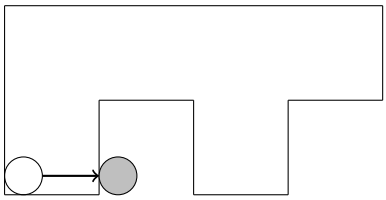

Рис. 3. Расположение второй фигуры способом «сквозного прохода»

за одной фигуры, и контур после расположения каждой из них изменяется. Таким образом, фигура движется вдоль контура контейнера, при нахождении подходящего места необходимо обновить контур с учетом расположения новой фигуры. На данный момент этот метод будет оставлен без особого внимания. Для его реализации необходимо создать цепь (растровую или же векторную) вдоль которой будет перемещаться фигура. Большую сложность, представляет в данном случае вопрос о выборе точки, относительно которой идёт движение.

4.2. Представление фигур в виде многоугольников

Представление фигур в виде многоугольников даёт хорошую точность аппроксимации. В таком представлении объем информации пропорционален числу вершин и не зависит от размера фигуры. Полигональное представление является первичным для фигур, а на его основе уже можно построить растровое представление, которое будет описано ниже.

Полигональный метод хоть и даёт высокую точность представления, но имеет очень высокую вычислительную сложность — $O(e^n)$, где n — число вершин фигуры.

Для того, чтобы проверить нет ли пересечений между какими-либо фигурами нужно выполнить следующий набор тестов [6]:

1. Проверить пересекаются ли описывающие прямоугольники фигур, если нет, то и фигуры не пересекаются, иначе перейти к следующему тесту.

2. Для каждой пары рёбер проверить, пересекаются ли их описывающие прямоугольники.

3. Проверить, пересекаются ли рёбра.

4. Проверить, лежат ли какие-либо вершины одного полигона, внутри другого.

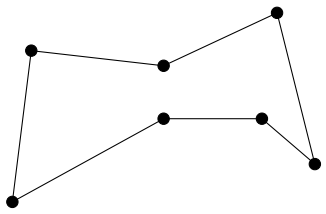

Рис. 4. Представление фигуры при помощи многоугольника

Проверка на пересечения текущей фигуры с ранее расположенными может выполняться различными способами. Первый вариант — через уравнение прямой с угловым коэффициентом. Этот метод для данной задачи будет излишним, ведь кроме проверки на пересечение двух отрезков, будут найдены точка пересечения и угловой коэффициент k данной прямой. Также придётся проверять специальные случаи, когда прямые параллельны, когда они направлены вертикально вверх. Лучше использовать метод проверки на основе псевдоскалярного произведения, ведь тогда сильно снизится вычислительная нагрузка. Данный метод подробно рассматривается в задачах вычислительной геометрии [4].

Для перемещения фигуры подходят только алгоритмы «тетрисного» движения. Применять «сквозное движение» мы не можем, так, как тогда придется постоянно проверять, не попала ли она внутрь другой, что в случае многоугольников сделать достаточно сложно. Для этого необходимо применять метод «трассировки луча» или приближённо считать комплексный интеграл, пользуясь интегральной формулой Коши [5].

Алгоритмы с представлением фигур в виде полигонов хорошо подходят для прямоугольников и несложных многоугольников с числом вершин до сотни. Для более сложных сильно возрастает время вычисления пересечений.

Проблема данного метода состоит в сложности обработки контура. Построить эквидистантный (равноудалённый во всех точках от исходного) контур, чтобы задать зазор — не такая уж лёгкая задача. Отделить внешний контур от внутренних тоже несколько сложнее, ведь контур может состоять из нескольких раздельных кривых, пусть и образующих в сумме одну замкнутую кривую.

4.3. Представление фигур в виде растровых матриц

Растровый метод, позволяет упростить геометрическую сложность фигуры и без дополнительных оптимизаций снизить сложность вычислений до $O(n^2)$.

Растровые методы предлагают разделить непрерывный раскройный лист на дискретные части, упрощая сложную геометрическую информацию до представления матрицей. Под матрицей будем понимать некоторое представление растра, в котором отмечены занятые и свободные места. Существуют различные методы представления.

Самый простой метод представления — это 1 для занятого деталью места и 0 для свободного. Раскраиваемый материал в данном случае представляется аналогично. На рис. 5 можно увидеть первичный вариант растрового представления [6]. В данном случае занятые области отмечены серым цветом.

Растровый способ не является первичным, он строится из фигуры, представленной многоугольником. Поэтому нельзя сразу говорить какие места заняты, а какие нет, ведь на входе имеется простое наложение многоугольника на растровую матрицу, что можно увидеть на рис. 6. Рассмотрим возможный алгоритм закраски занятых областей:

1. На первом шаге имеется исходный многоугольник и пустой растр, как показано на рис. 6. Выберем любое его ребро.

2. Так как каждое ребро задается координатами (x_1, y_1) и (x_2, y_2), то на втором шаге можно получить уравнение прямой с угловым коэффициентом, которая содержит данный отрезок. Для определённости будем считать, что $y_1 \leq y_2$.

3. Зададим множество $A = \{a_0, a_1, a_2, ..., a_{n-1}, a_n\}$, где $a_0 = y_1$, а $a_n = y_2$. Остальные элементы являются целыми числами из интервала (y_1, y_2).

4. Теперь, для каждой пары чисел $\{a_i, a_{i+1}\},\ i = 0 : n-1$ из имеющегося уравнения прямой $y = kx + b$, вычислим x_i и x_{i+1}.

5. Вычислив, все необходимые координаты, можно отметить занятыми (на изображениях закрашены серым цветом), все клетки на отрезке заданном координатами $(x_i, a_i),\ (x_{i+1}, a_i)$.

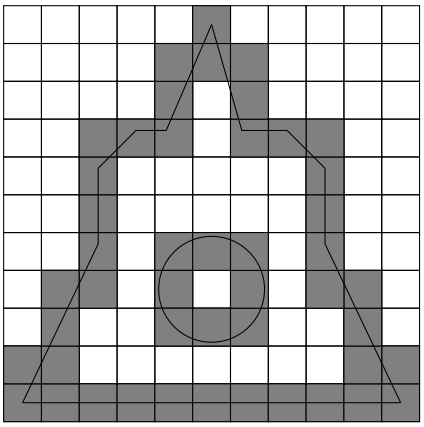

Рис. 5. Первичная растровая матрица

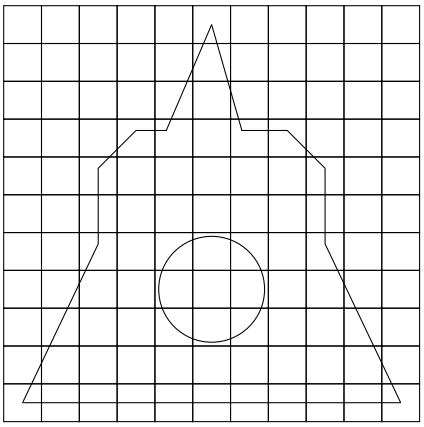

Рис. 6. Наложение многоугольника на растровую матрицу

Алгоритм рассматривает переход от отрезка к растру. Для перехода от многоугольника к растру достаточно повторить описанный алгоритм для всех его ребёр. В итоге получится то, что изображено на рис. 5.

Операция проверки на пересечения с другими фигурами, при текущем расположении в координате (x, y) фигуры с шириной равной w и высотой h, не требует никаких сложных действий. Производится простое суммирование по следующей формуле:

$$S = \sum_{i=1}^{h} \sum_{j=1}^{w} p_{i+y,j+x} f_{i,j}. \tag{1}$$

В данном случае, $p_{i+y,j+x}$ — значение в матрице раскройной плоскости со смещением на координату (x, y), а $f_{i,j}$ — значение в матрице фигуры. В случае если значение $S > 0$, то есть пересечение с некоторой фигурой, иначе его нет.

Как видно на рис. 5, не все области внутри фигуры отмечены как занятые. В таком случае, некоторые фигуры могут попасть внутрь других, что будет являться ошибкой. Данная проблема может быть решена различными способами. Самый простой способ — «залить» всё, что находится внутри внешнего конутура, отбрасывая пустоты внутри фигуры. Такой метод принесёт большие потери материала.

Рассмотрим метод, позволяющий учитывать пустые места внутри фигур. Для начала необходимо отделить внешний контур фигуры:

1. Примем за текущую самую левую занятую точку с наименьшей ординатой.

2. Отметим текущую точку, как точку контура (на изображениях будет отмечаться диагональной штриховкой).

3. Рассмотрим всех соседей текущей точки. Если есть соседняя точка, отмеченная цифрой 1, переходим в данную точку и возвращаемся на второй шаг. Если текущая точка — исходная, то закончить алгоритм, иначе повторить текущий шаг.

   Результат работы алгоритма можно увидеть на рис. 7.

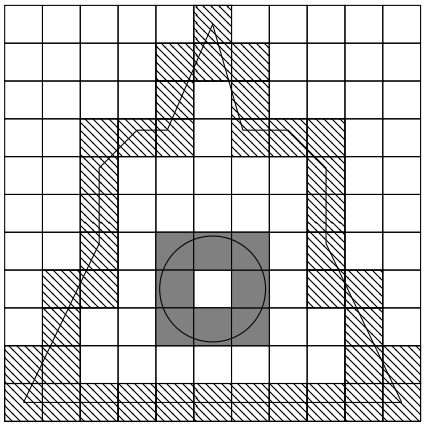

Рис. 7. Фигура с выделенным контуром

После отделения внешнего контура, необходимо отметить занятым, всё место между внешним контуром и контурами второго уровня. Это можно сделать следующим образом:

1. Первым шагом необходимо сохранить точки контура в отдельный масив и отметить их, как свободные (на изображениях не закрашены цветом) на растровой матрице.

2. Далее, выполняется «заливка». В процессе «заливки», свободные клетки отмечаются как временно занятые (на изображениях будет изображаться перекрёстной штриховкой). Результат первых двух шагов приведён на рис. 8.

3. Третьим шагом инвертируем «заливку». Для этого необходимо все временно занятые клетки надо отметить как своблдные. Свободные клетки необходимо отметить как временно занятые. Занятые клетки не затрагиваются. После этого можно вернуть на растровой матрице контур фигуры. Результат данного шага приведён на рис. 9.

4. После этого опять выполняется «заливка». Результат четвёртого шага приведён на рис. 10;

5. Последним шагом опять выполняется инверсия «заливки» по тем же правилам, что и на третьем шаге, только свободные клетки теперь отмечаются как занятые. Результат работы алгоритма приведён на рис. 11.

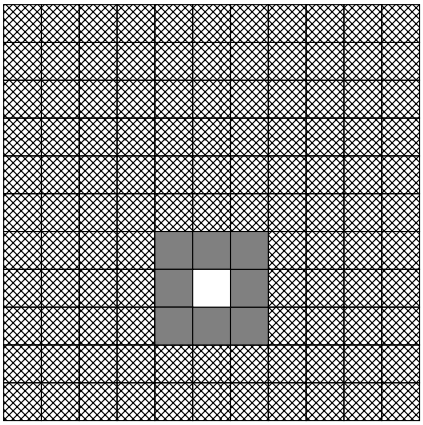

Рис. 8. Результат первых двух шагов алгоритма «заливки» пустых мест

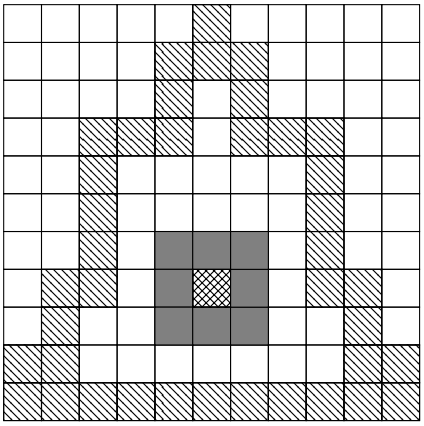

Рис. 9. Результат третьего шага алгоритма «заливки» пустых мест

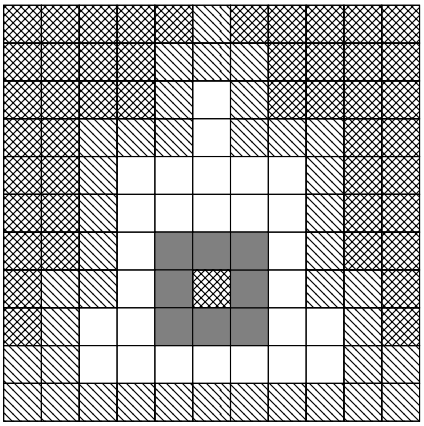

Рис. 10. Результат четвёртого шага алгоритма «заливки» пустых мест

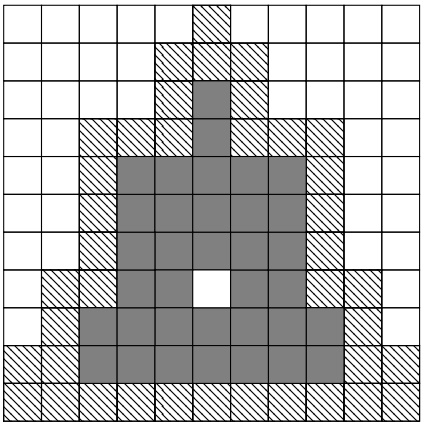

Рис. 11. Итоговый результат работы алгоритма «заливки» пустых мест

4.4. Условия корректности данных

Для корректности работы описанных выше алгоритмов и некоторых дальнейших выкладок необходимо ввести условия корректности.

Фигура является корректной, если выполняются следующие условия:

1. Фигура имеет замкнутый внешний контур.
2. Уровень вложенности контуров не более первого. Это означает, что контура могут вкладываться только во внешний контур, то есть если контур является вложенным во внешний, то в него уже не может быть вложен контур.
3. Вложенные контуры не должны иметь пересечений друг с другом.
4. Ни один из контуров не должен иметь самопересечений.

Несоблюдение приведённых выше условий может привести к непредсказуемым результатам работы алгоритма.

4.5. Уменьшение сложности до линейной

Для корректности дальнейших действий, введём условие, что движение фигур по плоскости материала происходит строго построчно. Из данного утверждения следует тот факт, что если фигура пересекает занятое другой фигурой место, то, хотя бы в одной из точек, пересечение будет лежать на контуре текущей фигуры.

Тогда можно снизить сложность поиска пересечений до $\mathrm{O}(n)$, где n — число точек, входящих в контур. Такой способ поиска пересечений является суммированием по следующей формуле:

$$S = \sum_{(i,j)\in\Gamma} p_{i+y,j+x}. \tag{2}$$

Данная формула аналогична формуле (1), но суммирование проходит только по тем точкам (j, i), которые входят во внешний контур Γ.

5. Принципы позиционирования фигур

Определившись с представлением фигур и методом их перемещения, разработчику необходимо решить, каким образом производить позиционирование. Первое с чего стоит начать — это поворот фигур. Некоторые авторы предлагают применять изменение угла методом дихотомии, что по сути является бинарным поиском. Но тут возникает большая проблема. Бинарный поиск работает на отсортированном множестве, а контур фигуры таковым не является, поэтому поворот следует выполнять просто с дискретным шагом изменения угла.

После того как фигура уже лежит где-то на плоскости, её положение надо оценить. Очень часто применяют принцип левого нижнего угла (ЛН-принцип). Так-как фигуры имеют сложный контур, можно предложить следующее улучшение данного метода. Пользуясь тем, что точки входящие контур известны, можно найти его центр масс по следующей формуле:

$$p_g = \sum_{(i,j)\in\Gamma} \frac{p_{i,j}}{n}, \tag{3}$$

где $p_{i,j}$ — это точка входящая в контур Г. Теперь можно не только пытаться расположить фигуру максимально влево и вниз, но и в случае, если два положения имеют одинаковую высоту, выбрать то, у которого центр масс фигуры ниже.

Рассмотрим на примере, как таким способом можно расположить некоторый пятиугольник. На рис. 12 видно, что пятиугольник расположился своим центром масс к низу, выставив наверх ту часть, которая оставляет больше свободного места для расположения оставшихся фигур.

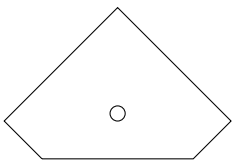

Рис. 12. Модифицированный ЛН-принцип

Можно придумать множество дальнейших модификаций данного метода, например, сначала оценивать описывающий прямоугольник множества

фигур и выбирать тот вариант расположения, при котором он меньше.

6. Генетические алгоритмы

Таким же образом, как нейронные сети пытаются имитировать мощь мозга, эволюционные алгоритмы (ЭА) воспроизводят биологическую эволюцию. Например, природа создала сложную структуру человеческого мозга из простейших элементов, пользуясь лишь мощью эволюции и естественного отбора. Так же как естественный отбор действует на популяции животных, позволяя лучшим выжить и передать свои гены потомкам, так же и ЭА действует, например, на последовательности раскроя, нейронные сети, электрические контуры и так далее. Они определяют качество каждого элемента в популяции, позволяя лучшим выжить и убивая слабейших. Далее — очень важный шаг — они позволяют различным решениям «скрещиваться» между собой, порождая, возможно, более жизнеспособные решения.

Генетический алгоритм (ГА) — самый популярный из эволюционных алгоритмов. Он был изобретён Джоном Холландом в Университете Мичигана в 1975 году. По изначальной задумке, ГА использовал только бинарные числа, но для извлечения большей пользы будем пользоваться десятичными числами. Сам алгоритм состоит из нескольких частей [7]:

1. Кодирование проблемы в виде генов.
2. Выведение первоначального поколения.
3. Вычисление оценок для индивидов.
4. Скрещивание.
5. Мутация.

Рассмотрим ГА на применительно к задаче раскроя [8].

6.1. Кодирование задачи

Основываясь на предложенных ранее методах представления и перемещения фигур, можно заметить, что не имеет смысла включать в геном

координаты положения и угол поворота так, как они будут вычислены при расположении последовательности. Достаточно кодировать лишь саму последовательность раскроя. Отметим, что для достижения результата, в геноме инидвида не должно быть повторов, соответственно каждый ген должен быть уникален. Таким образом индивидом будет являться — последовательность просто раскроя, ведь углы постановки и координаты будут восстановлены при расположении, а геном — порядковый номер фигуры в исходном наборе — это позволит избежать повторов.

6.2. Инициализация первого поколения

Первое поколение индивидов оказывает огромное влияние на результат всего ГА. Чтобы получить неплохой первичный набор, можно расположить по порядку отсортированные по площади фигуры, а остальных индивидов первого поколения получить с помощью мутации.

6.3. Оценка индивидов

Оценку результата можно проводить различными способами. Можно, к примеру, ввести некоторую оценочную функцию. Рассматривая задачу раскроя, за оценку можно взять занятую фигурами площадь. Далеко необязательно использовать какой-то один фактор, например, если индивид I характеризуется парой (n, h), где n — число размещённых фигур, а h — высота раскроя, то можно взять за оценку данную пару. Тогда то, что индивид I_1 лучше, чем индивид I_2 можно записать через следующее отношение ρ:

$$I_1 \ \rho \ I_2 = (n_1 > n_2 \vee (n_1 = n_2 \wedge h_1 < h_2)). \tag{4}$$

Тоже самое можно записать с помощью следующей функции оценки индивидов:

$$f(I) = n + h^{-1}, \tag{5}$$

тогда достаточно сравнивать числовые значения функций. Выбранная функция будет принимать большее значение при лучшем варианте раскроя. Вообще говоря, можно использовать любую функцию, исходя из условий задачи.

6.4. Скрещивание

Рассмотрим скрещивание на следующем примере:

1. Выберем двух индивидов, например [5 2 3 7 6 1 4] и [4 6 2 1 3 5 7].
2. Случайным образом выберем часть первого родителя, которая перейдёт в потомка: [5 2 (3 7) 6 1 4].
3. Удалим эти элементы из второго родителя: [4 6 2 1 (3) 5 (7)].
4. Первого потомка получим путём копирования генов второго родителя в первого: [4 6 3 7 2 1 5].
5. Повторим действия 1—3, поменяв родителей местами.
6. Второго потомка получим путём копирования генов первого родителя во второго: [5 3 2 1 7 6 4].

Известно, что в области определения функция может иметь не только абсолютные экстремумы, но и локальные [10]. поэтому в процессе скрещивания стоит использовать не только самых лучших, индивидов но и достаточно хороших в целях избежания попадания на точки локального минимума. Схождение к локальному экстремуму назвается преждевременной сходимостью или сходимостью к квазиоптимальному решению [9].

На рис. 13, для некоторой функции $f : X \to Y$, продемонстрированы точки квазиоптимального и оптимального решения. На рисунке цифра 1 — квазиоптимальное решение, а 2 — оптимальное.

6.5. Мутация

Мутация происходит после скрещивания и применяется к новым индивидам, на самом деле можно применять и к появившимся ранее. В случае раскроя в процессе мутации меняются местами два гена в последовательности. Вероятность мутации должна быть не очень высокой, приблизительно 10%, также можно изменять экспериментально. Иногда можно переставлять не просто два гена, а целые части последовательности, но не стоит делать этого слишком часто.

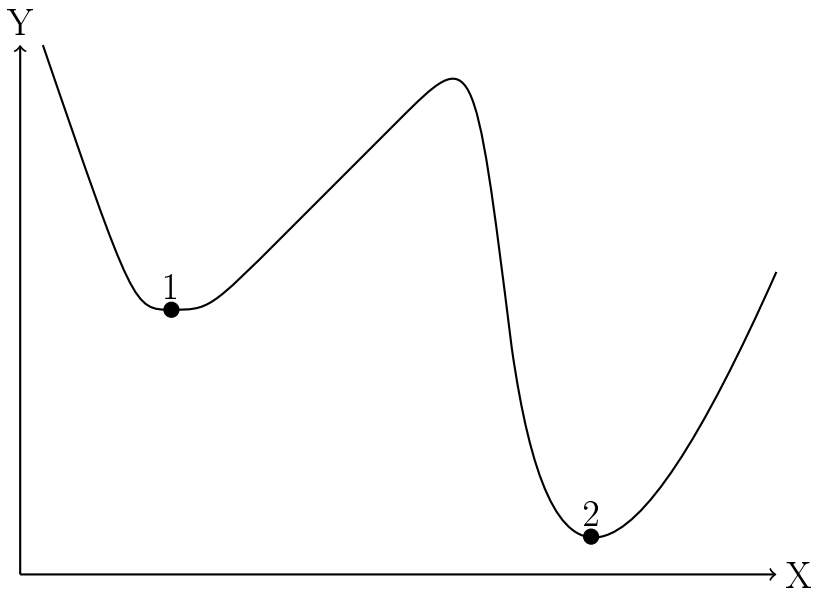

Рис. 13. Преждевременная сходимость

6.6. Отбор

Теперь, когда для каждого индивида вычислена оценка, можно провести отбор для создания нового поколения. Самый простой вариант — взять только те индивиды, у которых достаточно хорошая оценка. В классическом варианте генетического алгоритма, индивиды в новом поколении выбираются случайно, а вероятность попадания индивида в новое поколение будет пропорциональна его оценке. Например,

$$p_i = f_i^{-1} \sum f_k. \tag{6}$$

На самом деле, включать в новое поколение можно не только те индивиды, которые были выведены на данном шаге, но и те которые уже были ранее, ведь они могут быть не хуже чем те, что только что появились.

7. Реализация

Для написания программного кода был выбран язык программирования Go. Он обладает рядом преимуществ перед другими языками. Для полного понимания философии этого языка стоит начать с истории его возник-

новения.

Go был задуман в сентябре 2007 года Робертом Грисемером (Robert Griesemer), Робом Пайком (Rob Pike) и Кеном Томпсоном (Ken Thompson) из Google и анонсирован в ноябре 2009 года. Целью разработки было создание выразительного, высокоэффективного как при компиляции, так и при выполнении программ языка программирования, позволяющего легко и просто писать надежные высокоинтеллектуальные программы.

Go имеет поверхностное сходство с языком программирования C и обладает тем же духом инструментария для серьезных профессиональных программистов, предназначенного для достижения максимального эффекта с минимальными затратами. Но на самом деле Go — это нечто гораздо большее, чем просто современная версия языка программирования C. Он заимствует и приспосабливает для своих нужд хорошие идеи из многих других языков, избегая возможностей, которые могут привести к созданию сложного и ненадежного кода. Его способности к параллелизму новы и чрезвычайно эффективны, а подход к абстракции данных и объектно-ориентирован ному программированию непривычный, но необычайно гибкий. Как и все современные языки, Go обладает эффективным механизмом сбора мусора.

Go особенно хорошо подходит для инфраструктуры: построения инструментария и систем для работы других программистов. Однако, будучи в действительности языком общего назначения, он подходит для любого применения и становится все более популярным в качестве замены нетипизированных языков сценариев, обеспечивая компромисс между выразительностью и безопасностью. Программы Go обычно выполняются быстрее, чем программы, написанные на современных динамических языках, и не завершаются аварийно с неожиданными типами ошибок.

Go — это проект с открытым исходным кодом, так что исходные тексты его библиотек и инструментов, включая компилятор, находятся в открытом доступе. Свой вклад в язык, его библиотеки и инструментарий вносят многие программисты всего мира. Go работает на большом количестве Unix-подобных систем, таких как Linux, FreeBSD, OpenBSD, Mac OS X, а также на Plan 9 и Microsoft Windows; при этом программы, написанные для одной из этих сред, легко переносимы на другие.

7.1. Происхождение Go

Подобно биологическим видам, успешные языки порождают потомство, которое наследует наилучшие особенности своих предков. Скрещивание при этом иногда приводит к удивительным результатам. Аналогом мутаций служит появление радикально новых идей. Как и в случае с живыми существами, глядя на такое влияние предков, можно многое сказать о том, почему язык получился именно таким, какой он есть, и для каких условий работы он приспособлен более всего. На рис. 14 показано, какие языки повлияли на дизайн языка программирования Go [11].

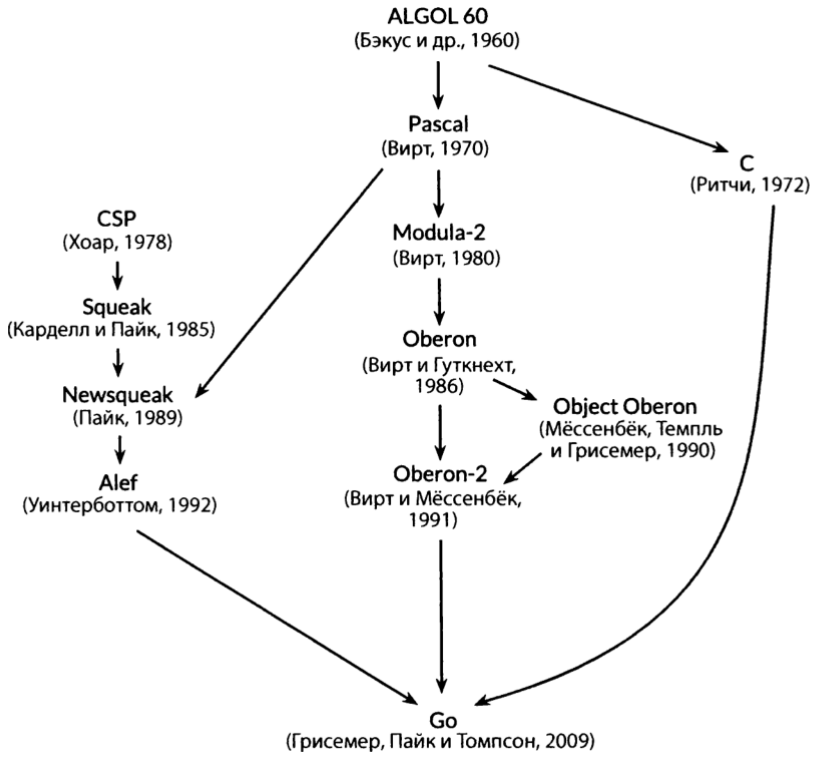

Рис. 14. Родственные связи Go с другими языками

7.2. Разраобка архитектуры и формата данных

Как сказано в [12]: «Есть два способа конструирования программного обеспечения. Один из них заключается в том, чтобы создавать такие простые программы, в которых нет недостатков; другой — в том, чтобы создавать программы настолько сложные, чтобы в них не было очевидных недостатков. Первый метод значительно труднее.»

Следуя принципам проектирования программного обеспечения для операционных систем семейства UNIX, модули тестирования API и само ядро реализующее алгоритм были разработаны раздельно, а для их взаимодействия был разработан текстовый формат обмена данными.

Важныи качеством модульного кода является инкапсуляция. Различные модули не должны зависить от реализации друг друга, а должны взаимодействовать с помощью программных интерфейсов приложений (API) — набора функций и структур данных, необходимых для работы модулей.

Для взаимодействия модулей тестирования и основного ядра был разработан текстовый протокол взаимодействия так, как текстовые потоки просты для чтения, записи и редактирования человеком без использования специальных средств. Текстовые протоколы являются более масштабируемыми чем бинарные потому, что в двоичных форматах для каждого значения определяется количество выделяемых битов. Данный протокол придерживается DSV-стиля (формат с разделителями значений). В первой строке группы значений записывается количество фигур и шаг поворота, далее идут точки примитивов. Каждая точка записываятся в одну строку, а координаты записаны через пробел. Группы точек между примитивами различных пустой строкой. Признаком конца фигуры, является доветочее, после неё начинается новая фигура. При проектировании формата учитывались основные соглашения Unix относительно текстовых форматов данных [13].

7.3. Структуры данных

Go является объектно-ориентированным языком программирования и позволяет разграничивать доступ к методам и полям объектов. C++ и подобные языки делают это с помощью модификаторов `private` и `public`. В языке Go принято использование camel case, и имена приватных полей и методов начинаются с прописной буквы, а публичных — с заглавной.

Рассмотрение структур данных следует начать с представления фигур в системе. Для этого используется три структуры. Первая, самая простая — `Point`, которая представляет точку в двумерном пространстве. Она имеет два открытых поля для: `X` и `Y` для представления координаты в пространстве.

```
type Point struct {
```

```
  X, Y float64
}
```

Листинг 1. Структура представления точки

Точки объединяются в примитивы с помощью структуры `Primitive` имеющей публичное поля типа `[]Points` — срез точек.

```
type Primitive struct {
  Points []Point
}
```

Листинг 2. Структура представления примитива

Из примитивов строится фигура, представленная структурой `Figure`. Поле `ID` необходимо для идентификации фигур, `Quant` — для указания количества повторений данной фигуры при построении множества фигур для раскроя, `Matrix` представляет собой матрицу афинного преобразования применённого к данной фигуре в процессе раскроя. Поля`Width` и `Height` — ширина и высота фигуры. Поле `AngleStep` представляет шаг угла поворота фигуры. `[]Primitive` — срез примитивов, входящих в данную фигуру. `MassCenter` — центр масс данной фигуры.

```
type Figure struct {
  ID, Quant                 int
  Matrix                    [][]float64
  Width, Height, AngleStep  float64
  Primitives                []Primitive
  MassCenter                Point
}
```

Листинг 3. Структура представления фигуры

Основной структурой для раскроя является `Rastr`, которая представляет растровую матрицу фигуры. Она состоит из полей: `RastrMatrix` — целочисленная растровая матрица, `Width` и `Height` — ширина и высота матрицы, `OuterContour` — целочисленные координаты точек внешнего контура фигуры в матрице. `PointInt` представлена также, как и `Point`, но её поля имеют тип `int`.

```
type Rastr struct {
  RastrMatrix    [][]int
  Width, Height  int
  OuterContour   []PointInt
}
```

Листинг 4. Структура представления растра

Информация о результате раскроя представляется структурой `Position`, состоящей из следующих полей: `Fig` — указатель на фигуру из задания (следует отметитить, что в Go указатели не являются таким мощным типом данных как в Си, в Go отсутствует указательная арифметика, и указатели являются просто ссылками на объект), `X` и `Y` — координаты размещения фигуры на плоскости, `Angle` — угол, на который была повёрнута фигура.

```
type Position struct {
  Fig          *Figure
  Angle, X, Y  float64
}
```

Листинг 5. Структура представления результата раскроя

Основная элемент генетического алгоритма индивид представлен структурой `Individual`, состоящей из трёх полей: `Height` — высота раскроя данного индивида, `Genom` — геном или раскройная последовательность, `Positions` — расположение фигур на раскройной плоскости.

```
type Individual struct {
  Height     float64
  Genom      []int
  Positions  []Position
}
```

Листинг 6. Структура представления индивида генетического алгоритма

7.4. Реализация основных функций и методов

Следует упомянуть некоторые идеи и особенности реализации методов и функций в Go.

Для многих других функций даже в хорошо написанных программах успех не гарантируется, потому что он зависит от факторов, находящихся вне контроля программиста. Например, любая функция, выполняющая ввод-вывод, должна учитывать возможность ошибки, и только наивный программист верит в то, что простое чтение или запись никогда не сбоит. Когда наиболее надежные операции неожиданно оказываются неудачными, обязательно необходимо знать, в чем дело.Таким образом, ошибки являются важной частью API пакета или пользовательского интерфейса приложения, и сбой является лишь одним из нескольких ожидаемых поведений программы. В этом заключается подход Go к обработке ошибок. Функция, для которой отказ является ожидаемым поведением, возвращает дополнительный результат, обычно последний. Если сбой имеет только одну возможную причину, результат представляет собой булево значение, обычно с именем ok. Чаще всего, и в особенности для ввода-вывода, неудачи могут иметь множество разнообразных причин, так что вызывающая функция должна знать, что именно произошло. В таких случаях тип дополнительного результата — error.

В Go нет перегрузки операторов и функций, а методы в Go объявляются с помощью вариации объявления обычных функций, в котором перед именем функции появляется дополнительный параметр. Этот параметр присоединяет функцию к типу этого параметра. Дополнительный параметр называется получателем метода приемника; это на звание унаследовано от ранних объектно-ориентированных языков, которые описывали вызов метода как “отправку сообщения объекту”. В Go не используется специальное имя, такое как this или self, для получателя; мы выбираем имя для получателя так же, как для любого другого параметра. Поскольку имя получателя будет использоваться часто, лучше выбрать что-то короткое и согласованно используемое во всех методах. В вызове метода аргумент получателя находится перед именем метода [11].

Рассмотрим реализацию функции мутации генетического алгоритма, она является методам типа `Individual`. В результате она возвращает кортеж из нового индивида и ошибки, если таковая была. Едиственная аварийная ситуация для данной функции — слишком маленький геном индивида. Во время выполнения функция копирует геном входного индивида в новый

индивид и генерирует два различных случайных числа с индексами массива, затем она меняет местами гены на этих номерах.

```
func (indiv *Individual) Mutate() (*Individual, error)
  {
  if len(indiv.Genom) < 2 {
    return nil, errors.New("Too short genom")
  }

  mutant := new(Individual)
  genomSize := len(indiv.Genom)
  mutant.Genom = make([]int, genomSize)
  copy(mutant.Genom, indiv.Genom)
  i := rand.Int() % genomSize
  j := rand.Int() % genomSize
  for i == j {
    j = rand.Int() % genomSize
  }
  mutant.Genom[i], mutant.Genom[j] = mutant.Genom[j],
    mutant.Genom[i]
  return mutant, nil
}
```

Листинг 7. Метод для реализации мутации индивида

Скрещевание индивидов представлено функцией, принимающей на вход указатели на индивидов-родителей. Также как и мутация, она возвращает нового индивида и сообщение об ошибке, если таковая была. Для этой функции возможно две аварийных ситуации: индивиды имели различные длины геномов или же геномы одинаковые, но слишком короткие. После проверки на ошибки, функция генерирует два случайных числа с номерами генов, которые будут взяты из первого родителя и перейдут в ребёнка, далее свободные места заполняются генами второго родителя.

```
func Crossover(parent1, parent2 *Individual)
  (*Individual, error) {
  genSize1 := len(parent1.Genom)
```

```
genSize2 := len(parent2.Genom)
if genSize1 != genSize2 {
  return nil, errors.New("Different sizes of genoms")
}

if genSize1 < 3 {
  return nil, errors.New("Too short genom")
}

g1 := rand.Int() % genSize1
g2 := rand.Int() % genSize2

child := new(Individual)
child.Genom = make([]int, genSize1)
child.Genom[g1] = parent1.Genom[g1]
child.Genom[g2] = parent1.Genom[g2]

for i, j := 0, 0; i < genSize2 && j < genSize2; i, j
  = i+1, j+1 {
  if j == g1 || j == g2 {
    i--
    continue
  }

  if parent2.Genom[i] == child.Genom[g1] ||
     parent2.Genom[i] == child.Genom[g2] {
    j--
    continue
  }

  child.Genom[j] = parent2.Genom[i]
}
```

```
  return child, nil
}
```

Листинг 8. Функция для реализации скрещевания индивидов

Следующая функция выполняет преобразование растра в фигуру. первый аргумент представляет параметр для масштабирования растра, данное преобразование позволяет за счёт точности увеличить скорость раскроя. Второй аргумент позволяет задать увеличение границы, это необходимо для того, чтобы учесть толщину режущего элемента. В цикле функции выполняется описанный выше алгоритм перехода, для каждого отрезка, входящего в фигуру. Функции заливки, масштабирования и увеличения толщины границ также являются особенностью конкретной реализации и рассматриваться не будут.

```
func (fig *Figure) figToRastr(resize int, bound int)
  *Rastr {
  rastr := new(Rastr)

  rastr.Width = int(fig.Width) + 1
  rastr.Height = int(fig.Height) + 1
  rastr.OuterContour = make([]PointInt, 0,
    rastr.Width*rastr.Height)
  rastr.RastrMatrix = make([][]int, rastr.Height)
  for i := 0; i < rastr.Height; i++ {
    rastr.RastrMatrix[i] = make([]int, rastr.Width)
  }

  for i := 0; i < len(fig.Primitives); i++ {
    for j := 0; j < len(fig.Primitives[i].Points)-1;
      j++ {
      var top, bottom Point

      if fig.Primitives[i].Points[j].Y >
        fig.Primitives[i].Points[j+1].Y {
        top = fig.Primitives[i].Points[j]
        bottom = fig.Primitives[i].Points[j+1]
```

```
} else {
  top = fig.Primitives[i].Points[j+1]
  bottom = fig.Primitives[i].Points[j]
}

intervals := getIntervals(bottom.Y, top.Y)
if top.Y-bottom.Y > 1.0 {
  for k := 0; k < len(intervals)-1; k++ {
    x1 := calcX(top, bottom, intervals[k])
    x2 := calcX(top, bottom, intervals[k+1])
    y := intervals[k]
    // y1 := intervals[k+1]

    step := 1.0
    if x2 <= x1 {
      step = -1.0
    }
    rastr.RastrMatrix[int(y)][int(x1)] = filled
    rastr.RastrMatrix[int(y)][int(x2)] = filled
    for x := math.Trunc(x1); x !=
      math.Trunc(x2); x += step {
      rastr.RastrMatrix[int(y)][int(x)] = filled
    }
  }
} else {
  x1 := bottom.X
  x2 := top.X
  y := bottom.Y

  step := 1.0
  if x2 <= x1 {
    step = -1.0
  }
```

```
      rastr.RastrMatrix[int(y)][int(x1)] = filled
      rastr.RastrMatrix[int(y)][int(x2)] = filled
      for x := math.Trunc(x1); x != math.Trunc(x2);
        x += step {
        rastr.RastrMatrix[int(y)][int(x)] = filled
      }
    }
  }
}

if bound > 0 {
  rastr = makeBound(rastr, bound)
}

if resize > 0 {
  rastr = resizeRastr(rastr, resize)
}

rastr.findContour()
for i := 0; i < rastr.Height; i++ {
  for j := 0; j < rastr.Width; j++ {
    if rastr.RastrMatrix[i][j] == contour {
      rastr.OuterContour =
        append(rastr.OuterContour, pointIntNew(j,
        i))
    }
  }
}

rastr.floodRastr()

return rastr
```

```
}
```

Листинг 9. Метод для реализации перехода от фигуры к растру

Рассмотрим функцию для раскроя заданной последовательности фигур. Данная функция завершится аварийно если были переданы некорректные параметры: неположительная ширина или высота раскройной плоскости, отрицательная толщина границы или параметр масштабирования. В последовательности фигур могут встречатся фигуры с одинаковым `ID`, как только не нашлось места для одной из таких фигур, все следующие пропускаются, такой подход уменьшает время выполнения. Если на вход пришёл индивид с нулевой длиной — это значит, что это генерация первичного решения и будут располагаться только фигуры из заданного массива, если же длина генома ненулевая, сначала располагаются фигуры из генома, а затем те из входного массива, которые ещё не были использованы в геноме.

```
for len(figSet) > 0 {
    indivs := make([]*gonest.Individual, 1)
    indivs[0] = new(gonest.Individual)
    err = gonest.RastrNest(figSet, indivs[0], width,
      height, bound, resize)
    if err != nil {
      log.Fatal("Error! RastrNest: ", err)
    }

    for i := 0; i < iterations; i++ {
      nmbNew := 0
      oldLen := len(indivs)
      wg := new(sync.WaitGroup)
      for j := 0; j < oldLen-1 && indivs[j+1].Height
        != math.Inf(1) &&
        nmbNew < maxThreads; j++ {
        var children [2]*gonest.Individual

        children[0], err = gonest.Crossover(indivs[j],
          indivs[j+1])
```

```
    if err != nil {
      log.Println(err)
      break
    }
    children[1], _ = gonest.Crossover(indivs[j+1],
      indivs[j])

    for k := 0; k < 2; k++ {
      equal := false
      for m := 0; m < oldLen+nmbNew; m++ {
        if gonest.IndividualsEqual(indivs[m],
          children[k], figSet) {
          equal = true
          break
        }
      }

      if !equal {
        nmbNew++
        wg.Add(1)
        go nestRoutine(children[k], wg)
        indivs = append(indivs, children[k])
      }
    }
  }

  for j := 0; j < maxMutateTries && nmbNew <
    maxThreads; j++ {
    mutant, err := indivs[0].Mutate()
    if err != nil {
      break
    }
```

```
      equal := false
      for k := 0; k < oldLen+nmbNew; k++ {
        if gonest.IndividualsEqual(indivs[k],
          mutant, figSet) {
          equal = true
          break
        }
      }

      if !equal {
        nmbNew++
        wg.Add(1)
        go nestRoutine(mutant, wg)
        indivs = append(indivs, mutant)
      }
    }

    wg.Wait()
    sort.Sort(gonest.Individuals(indivs))
  }

  err = gonest.RastrNest(figSet, indivs[0], width,
    height, bound, resize)
  for i := 0; i < len(indivs[0].Positions); i++ {
    a := indivs[0].Positions[i].Fig.Matrix[0][0]
    b := indivs[0].Positions[i].Fig.Matrix[1][0]
    c := indivs[0].Positions[i].Fig.Matrix[0][1]
    d := indivs[0].Positions[i].Fig.Matrix[1][1]
    e := indivs[0].Positions[i].Fig.Matrix[0][2]
    f := indivs[0].Positions[i].Fig.Matrix[1][2]

    fmt.Printf("%d\n", indivs[0].Positions[i].Fig.ID)
    fmt.Printf("matrix(%f, %f, %f, %f, %f, %f)\n",
```

```
            a, b, c, d, e, f)
        }
        fmt.Println(":")
        newFigSet := make([]*gonest.Figure, 0)
        for i := 0; i < len(figSet); i++ {
          var j int
          found := false
          for j = 0; j < len(indivs[0].Genom); j++ {
            if i == indivs[0].Genom[j] {
              found = true
              break
            }
          }

          if !found {
            newFigSet = append(newFigSet, figSet[i])
          }
        }

        figSet = newFigSet
    }
```

Листинг 10. Функция для раскроя заданной последовательности

Ниже описана функция, для расположения фигуры на плоскости. На вход она принимает указатель на саму фигуру, массив уже расположенных фигур и параметры описанные ранее. Функция работает с копией фигуры и не модифицирует фигуры в пользовательском массиве. В цикле с заданным шагом кгла поворта, выполняется переход от фигуры к растру для фигуры и с шагом в один пиксель происходит проверка расположения фигуры. Расположение фигуры делается сначала на первое подходящее место, затем отыскивыются лучшие места расположения исходя из функции оценки, являющейся особенностью реализации. Если было найдено подходящее расположение фигуры, её растр сохраняется в раскройную плоскость, а вызывающей функции сообщается результат операции.

```
func placeFigHeight(fig *Figure, posits *[]Position,
  width, height, resize, bound int,
  place [][]int) bool {
  placed := false

  for angle := 0.0; angle < 360.0; angle +=
    fig.AngleStep {
    currFig := fig.copy()
    currFig.Rotate(angle)
    rastr := currFig.figToRastr(resize, bound)
    if rastr.Width > width/resize || rastr.Height >
      height/resize {
      return false
    }
    for y := 0; y < height-rastr.Height; y++ {
      for x := 0; x < width-rastr.Width; x++ {
        cross := false

        for k := 0; k < len(rastr.OuterContour); k++ {
          i, j := rastr.OuterContour[k].Y,
            rastr.OuterContour[k].X

          if place[y+i][x+j] > 0 {
            cross = true
            break
          }
        }

        if cross {
          continue
        }

        if checkPositionHeight(currFig, posits,
```

```
            float64(x*resize), float64(y*resize),
            float64(width), float64(height), &placed) {
            (*posits)[len(*posits)-1].Angle = angle
          }

          x = width
          y = height
        }
      }
    }

    if !placed {
      return false
    }

    rastr :=
      (*posits)[len(*posits)-1].Fig.figToRastr(resize,
      bound)
    for i := 0; i < rastr.Height; i++ {
      for j := 0; j < rastr.Width; j++ {
        x := int((*posits)[len(*posits)-1].X) / resize
        y := int((*posits)[len(*posits)-1].Y) / resize
        place[i+y][j+x] += rastr.RastrMatrix[i][j]
      }
    }

    return true
}
```

Листинг 11. Функция расположения фигуры на плоскости

Рассмотрим основной цикл программы. Выполнение происходит пока во входном наборе есть фигуры. Далее создаётся первоначальное решение, затем в цикле с заданным количеством итераций генетического алгоритма, происходит улучшение базисного решения. Сначала, если есть возможность,

происходит операция скрещевания, и для новых индивидов сразу запускается функция раскроя в отдельном потоке. Далее, пока не будет набрано необходимое количество новых индивидов, либо не истечёт число попыток, будет происходить операция мутации. После выполнения цикла генетического алгоритма, на стандартный вывод подаётся информация о расположенных фигурах и отдельный модуль сохраняет результат в файл, далее все расположенные фигуры удаляются из массива и цикл повторяется с генерации базисного решения.

```
for len(figSet) > 0 {
    indivs := make([]*gonest.Individual, 1)
    indivs[0] = new(gonest.Individual)
    err = gonest.RastrNest(figSet, indivs[0], width,
      height, bound, resize, rastrType,
      placementMode)
    if err != nil {
      log.Fatal("Error! RastrNest: ", err)
    }

    for i := 0; i < iterations; i++ {
      // fmt.Println("ITERATION ", i)
      for j := 0; j < len(indivs); j++ {
        log.Printf("len=%v height=%v genom=%v\n",
          len(indivs[j].Genom),
          indivs[j].Height, indivs[j].Genom)
      }

      nmbNew := 0
      oldLen := len(indivs)
      wg := new(sync.WaitGroup)
      for j := 0; j < oldLen-1 && indivs[j+1].Height
        != math.Inf(1) &&
        nmbNew < maxThreads; j++ {
        var children [2]*gonest.Individual
```

```
    children[0], err = gonest.Crossover(indivs[j],
      indivs[j+1])
    if err != nil {
      log.Println(err)
      break
    }
    children[1], _ = gonest.Crossover(indivs[j+1],
      indivs[j])

    for k := 0; k < 2; k++ {
      equal := false
      for m := 0; m < oldLen+nmbNew; m++ {
        if gonest.IndividualsEqual(indivs[m],
          children[k], figSet) {
          equal = true
          break
        }
      }

      if !equal {
        nmbNew++
        wg.Add(1)
        go nestRoutine(children[k], wg)
        indivs = append(indivs, children[k])
      }
    }
  }

  for j := 0; j < maxMutateTries && nmbNew <
    maxThreads; j++ {
    mutant, err := indivs[0].Mutate()
    if err != nil {
```

```
        break
      }

      equal := false
      for k := 0; k < oldLen+nmbNew; k++ {
        if gonest.IndividualsEqual(indivs[k],
          mutant, figSet) {
          equal = true
          break
        }
      }

      if !equal {
        nmbNew++
        wg.Add(1)
        go nestRoutine(mutant, wg)
        indivs = append(indivs, mutant)
      }
    }

    wg.Wait()
    sort.Sort(gonest.Individuals(indivs))
  }

  err = gonest.RastrNest(figSet, indivs[0], width,
    height, bound, resize, rastrType,
    placementMode)
  // a = append(a[:i], a[i+1:]...)
  for i := 0; i < len(indivs[0].Positions); i++ {
    a := indivs[0].Positions[i].Fig.Matrix[0][0]
    b := indivs[0].Positions[i].Fig.Matrix[1][0]
    c := indivs[0].Positions[i].Fig.Matrix[0][1]
    d := indivs[0].Positions[i].Fig.Matrix[1][1]
```

```
        e := indivs[0].Positions[i].Fig.Matrix[0][2]
        f := indivs[0].Positions[i].Fig.Matrix[1][2]

        fmt.Printf("%d\n", indivs[0].Positions[i].Fig.ID)
        fmt.Printf("matrix(%f, %f, %f, %f, %f, %f)\n",
          a, b, c, d, e, f)
      }
      fmt.Println(":")
      newFigSet := make([]*gonest.Figure, 0)
      for i := 0; i < len(figSet); i++ {
        var j int
        found := false
        for j = 0; j < len(indivs[0].Genom); j++ {
          if i == indivs[0].Genom[j] {
            found = true
            break
          }
        }

        if !found {
          newFigSet = append(newFigSet, figSet[i])
        }
      }

      figSet = newFigSet
    }

}
```

Листинг 12. Цикл основной программы

8. Численный эксперимент

Прежде чем переходить к рассмотрению входных данных, следует отметить, что для удобства работы все расчёты в программе производятся в пикселях (пкс), 1 мм = 3,543307 пкс [14].

Для проверки работоспособности и корректности алгоритма были проведены численные эксперименты на двух различных наборах данных. В обоих наборах заданы нулевое масштабирование и нулевая граница. Рассмотрим постановку задачи для первого набора данных. Имеется входной лист 1000×1000 пкс, за 10 итераций необходимо отыскать наиболее оптимальное расположение следующего набора фигур на листе:

1. «Олень» — 5 штук, с шагом 15 градусов.
2. «Обезьянка» — 5 штук, с шагом 15 градусов.
3. «Голубь» — 5 штук, с шагом 15 градусов.
4. «Лошадка» — 5 штук, с шагом 15 градусов.
5. «Ангелок» — 5 штук, с шагом 15 градусов.

Первичное решение строилось с помощью расположения фигур по убыванию площади, в него вошло 14 фигур, занятая высота на листе составила $947,115$ пкс. В результате работы генетического алгоритма из первичного индивида был получен индивид, содержащий 15 фигур с занятой высотой на листе $948,115$ пкс. Увеличение высоты в данном случае нельзя считать ухудшением результата так, как была расположена дополнительная фигура. Остаток из 10 фигур полностью расположился на втором раскройном листе. Высота первичного индивида для остатка фигур составила $982,065$ пкс, в результате было достигнуто улучшение до $930,909$ пкс. Результат работы для данного набора приведён на рис. 15.

Теперь рассмотрим постановку задачи для второго набора данных. Имеется входной лист 1500×1500 пкс, за 10 итераций необходимо отыскать наиболее оптимальное расположение следующего набора фигур на листе:

1. «Кольцо» — 10 штук, с шагом 15 градусов.

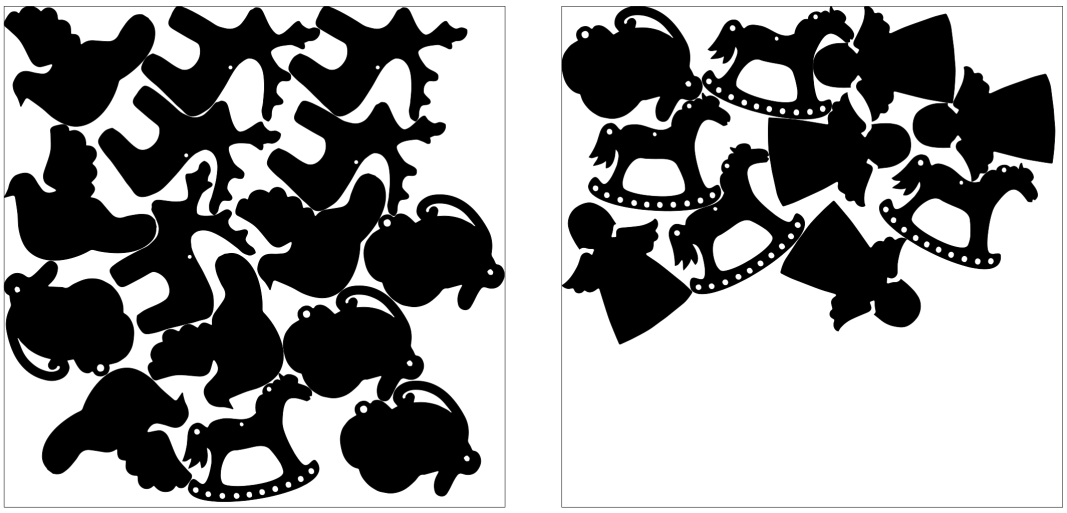

Рис. 15. Первый и второй раскройные листы первого набора фигур

2. «Рамка» — 10 штук, с шагом 15 градусов.

3. «Треугольник» — 10 штук, с шагом 15 градусов.

Первичное решение так же, как и для первого набора, строилось с помощью расположения фигур по убыванию площади, в него вошли все 3— фигур, занятая высота на листе составила $1388,258$ пкс. В результате работы алгоритма улучшения по высоте достигнуто не было, но данный набор демонстрирует возможность важную возможность алгоритма — расположение фигуры в фигуре. Резульат работы приведён на рис. 16.

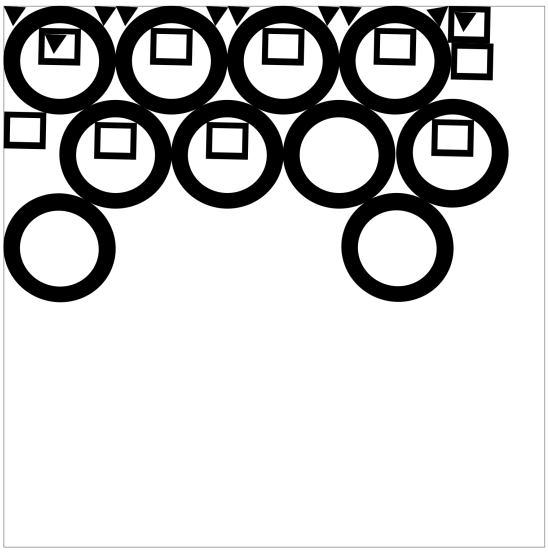

Рис. 16. Раскройный лист второго набора фигур

9. Заключение

В процессе исследования было реализовано API на языке программирования Go для разработки систем автоматизированного раскроя. Сравнение алгоритмов и на основе полигонов и растров показало следующие преимущества раскроя с помощью растровых матриц:

— простота реализации;

— масштабируемость алгоритма;

— более высокая скорость работы при достаточно малой потере точности;

— простой поиск свободных мест внутри других фигур.

Следует отметить чувствительность генетического алгоритма к первичному индивиду, даже различная сортировка фигур по площади приведёт к получению двух сильно отличающихся решений. Но, в отличие от человека, одну и ту же последовательность один алгоритм будет кроить одинаково. Для получения различных вариантов раскроя одной последовательности необходимо придумывать различные функции расположения и оценки полученного расположения.

Сравнивая разработку на языках C и Go, необходимо отметить значительное упрощение процесса при использовании языка Go. Хотя Go и несколько уступает C в скорости, но не на столько, чтобы этото стало минусом.

В дальнейшем возможно переписать с Go на Rust, имеющий скорость C и безопасность Java. Единственный минус Rust — высокий порог вхождения. Также возможно добавление новых функций оценки и движения.

Список литературы

[1] Dyckhoff H. A typology of cutting and packing problems // European Journal of Operational Research. № 44. 150—152 p.

[2] Залгаллер В. А., Канторович Л. В. Рациональный раскрой промышленных материалов. Новосибирск: Наука, 1971.

[3] Никитенков В. Л., Холопов А. А. Задачи линейного программирования и методы их решения. Сыктывкар: Издательство Сыктывкарского университета, 2008. 143 с.

[4] Прасолов В. В. Задачи по планиметрии. — 4-е изд., дополненное. М.: МЦНМО, 2001. 584 с

[5] Шабат Б. В. Введение в комплексный анализ. М.: Наука, 1969. 91 с.

[6] Benell A. J., Olivera F. J. The geometry of nesting problems: A tutorial // European Journal of Operational Research. 2008. № 184. 399—402 p.

[7] MacLeod C. An Introduction to Practical Neural Networks and Genetic Algorithms For Engineers and Scientists. 85 p.

[8] He Y., Liu H. Algorithm for 2D irregular-shaped nesting problem based on the NFP algorithm and lowest gravity-center principle // Journal of Zhejiang University. 2006. № 7. 571 — 574 p.

[9] Панченко Т. В. Генетические Алгоритмы; под ред. Ю. Ю. Тарасевича. Издательский дом «Астраханский университет». 2007. 16 с.

[10] Кудрявцев Л. Д. Математический анализ. — 2-е изд. М.: Высшая школа, 1973. — Т. 1.

[11] Донаван А. А. А., Керниган Б. У. Язык программировния Go. М., СПб., Киев: Вильямс, 2016. 11—12 с., 160—161 с., 191—192 с.

[12] Хоар Ч. А. Р. The Emperor's Old Clothes. CACM, 1981.

[13] Реймонд Э. С. Искусство программирования для UNIX. CACM, М., СПб., Киев: Вильямс, 2005.

[14] Coordinate Systems, Transformations and Units [Электронный ресурс] / W3C. 6 мая 2017. URL: https://www.w3.org/TR/SVG/coords.html.