

УДК 519.856

ГЕНЕТИЧЕСКИЙ АЛГОРИТМ С ДИНАМИЧЕСКИМ РАСПРЕДЕЛЕНИЕМ ВЕРОЯТНОСТЕЙ ВЫБОРА ГЕНЕТИЧЕСКИХ ОПЕРАТОРОВ ДЛЯ РЕШЕНИЯ ЗАДАЧ С ЦЕЛОЧИСЛЕННЫМ КОДИРОВАНИЕМ ГЕНОВ

Ю. О. Кашкарева

Челябинский государственный университет, Челябинск, Россия
kashkareva.julija@gmail.com

Разработаны и протестированы два генетических алгоритма построения суперстроки для произвольного бинарного файла. Они основаны на адаптивном подборе используемого набора генетических операторов.

Ключевые слова: построение суперстроки, генетические алгоритмы, задача коммивояжёра.

Введение

Классические генетические алгоритмы (ГА) работают с хромосомами в виде битовых строк, которые можно разорвать в произвольном месте, скрестить и получить допустимое решение. Большинство модификаций генетических алгоритмов так же оперируют битовыми строками. В частности, классические функции для тестирования эффективности ГА, такие как функции Растригина или Де Джонга, предполагают кодирование решений для поиска оптимума в виде бинарной строки.

Однако существует класс задач, для которых представление решений в указанном виде неэффективно или невозможно. К таким задачам, в частности, относится задача коммивояжёра (ЗК). Одним из методов решения ЗК является кодирование потенциальных решений с помощью последовательности целых чисел, указывающей порядок обхода графа. Хромосома для такого способа кодирования представляется в виде массива целых чисел, а каждое число представляет из себя отдельный ген, который наследуется потомком целиком. При таком способе кодирования хромосом, в свою очередь, меняются генетические операторы (ГО), такие как скрещивание и мутация, в них появляется понятие «ген».

В своей работе мы изучали применение динамического генетического алгоритма для сжатия и обфускации файлов с помощью решения задачи приближённого поиска кратчайшей суперстроки (Shortest Common Superstring, SCS), в конечном счёте сводящегося к решению асимметричной задачи коммивояжёра (Asymmetric Traveling Salesman Problem, ATSP).

1. Задача поиска кратчайшей суперстроки

Пусть даны строки $x = x_1 \dots x_n$ и $y = y_1 \dots y_m$. Строка y называется подстрокой x , если существует $i \in [0, n - m]$, такое, что $y_j = x_{i+j}$, $1 \leq j \leq m$. В этом случае x является суперстрокой для y . Задача поиска кратчайшей суперстроки заключается в следующем: дано множество из N строк $S = [s_1 \dots s_N]$ над конечным алфавитом

Σ , требуется найти кратчайшую суперстроку s , которая содержит каждую строку s_i как подстроку.

В известных нам работах всегда предполагалось, что исходное разбиение на подстроки задано заранее и не является частью алгоритма построения суперстроки. Улучшения в алгоритме поиска суперстроки были связаны именно с определёнными характеристиками исходных подстрок.

Мы предположили, что следует изучить возможность получения оптимального разбиения на подстроки. При этом мы предположили, что чем на большее число подстрок мы разобьём исходный текст и чем меньшей длины они будут, тем более короткую суперстроку мы сможем гипотетически получить на их основе, потому что добиться совпадения коротких подстрок проще, нежели длинных. Так, при подстроках длиной в один символ суперстрока совпадёт по длине с алфавитом. Однако разбиение исходного текста на слишком мелкие фрагменты приводит к усложнению вычисления непосредственно SCS, так как сложность задачи определяется именно количеством подстрок. Так же растут накладные расходы на восстановление исходной последовательности подстрок. Очевидно, что параметры разбиения будут сильно зависеть от исходного файла. Таким образом, мы имеем задачу, для которой известен алгоритм проверки конкретного решения, однако затруднён поиск всех возможных решений. Для таких задач хорошо подходят эвристические алгоритмы, в частности ГА.

На производительность ГА и качество найденных им решений существенно влияют параметры работы алгоритма и набор использованных генетических операторов [1]. Поскольку разрабатываемый нами алгоритм не является типовым, то требуется установить, какие генетические операторы будут давать наилучшие результаты. При этом поскольку мы будем получать вероятностное решение при построении разбиения и по нему строить суперстроку, то важно также, чтобы эти алгоритмы хорошо работали вместе.

В статье Ю. А. Бюргера [2] предложена идея динамической самоорганизации параметров ГА. На основе этой идеи некоторые авторы разработали алгоритмы для решения разнообразных задач. В частности, в работе Ю. Ю. Петрова [3] создан ГА с регуляцией вероятностей для решения задачи упаковки в контейнеры, а в работе А. С. Мясникова [4] — ГА с динамическим распределением вероятностей выбора генетических операторов с оптимизацией по времени работы алгоритма. В данных работах алгоритм применялся для ГО с классическим кодированием хромосом в виде битовых строк. При эмпирической проверке разработанные алгоритмы давали положительные результаты.

Указанные алгоритмы обладают рядом достоинств по сравнению с классическими алгоритмами. В первую очередь данные алгоритмы могут быть непосредственно применены к любой задаче, и в ходе их работы произойдёт настройка выбора ГО, дающих наилучшее решение для задачи с заданными параметрами и указанными начальными данными. Во-вторых, в ходе работы данных алгоритмов можно собрать статистику об использовании ГО и затем применить схему простого ГА с указанными генетическими операторами.

На базе указанных алгоритмов нами были разработаны две собственные версии ГА с динамическим распределением вероятностей выбора генетических операторов (далее — ГА с ДРВ): одна на базе простого ГА (далее ПГА), другая — по аналогии с алгоритмом, описанным в [4], однако без оптимизации по времени.

После того как мы некоторым образом определим разбиение на подстроки исходного файла, мы можем приступить к непосредственному поиску точного и при-

ближённому решению задачи SCS.

Поиск SCS — это NP-полная задача [5]. В [6] показано, что данная задача относится к классу MAX-SNP и отсутствует полиномиальный алгоритм, который позволял бы найти решение для задачи SCS, отличающееся от оптимального на произвольную, но заранее заданную константу. Задача поиска кратчайшей строки сводится к асимметричной задаче коммивояжёра (ATSP), и при числе подстрок, равном N , точное решение может быть найдено за время 2^N с использованием полиномиальной памяти [7]. Таким образом, точное решение данной задачи может быть найдено только при небольшом числе подстрок. Обычно для поиска точного решения применяют алгоритм Литтла [8].

В случае большей размерности задачи ищут её приближённое решение с помощью аппроксимационных алгоритмов, имеющих полиномиальное время работы, и эвристических алгоритмов. Лучший известный полиномиальный алгоритм SCS имеет фактор приближения 2.5 [9] и основывается на сведении SCS к ATSP, после чего выполняется построение циклического покрытия минимальной длины. Данный алгоритм достаточно сложен и, согласно проанализированным источникам, не используется на практике. Самый известный и наиболее широко распространённый полиномиальный алгоритм решения получил название жадного (GREEDY) и используется для нахождения решений с фактором приближения 4.

Эвристические, в частности генетические, алгоритмы работают гораздо дольше, нежели аппроксимационные полиномиальные, однако дают результат быстрее, чем алгоритмы точного решения задачи, и получаемая ими суперстрока может быть существенно короче.

В известных нам работах, описывающих использование ГА для решения задачи поиска SCS, применялись ПГА, а также некоторые его адаптивные улучшения, однако не обосновывался выбор определённых ГО. Даже в распределённых вариантах по типу островного алгоритма [10; 11] не предлагалось использовать различные ГА. В целом же было показано, что во многих случаях ГА действительно эффективно ищут приближённое решение для SCS. Мы применили полученные нами ГА с ДРВ для решения данной задачи и анализа использования различных ГО. При этом из-за особенностей задачи коммивояжёра набор ГО был соответствующим образом изменён.

2. Описание ГО для алгоритма разбиения файла на подстроки

2.1. Отличие рассматриваемых алгоритмов от классической схемы

В наших ГА хромосома представляется не битовой строкой, как в классическом алгоритме, а набором генов. На первом этапе для кодирования разбиения исходного файла на подстроки один ген будет представлять собой большое целое число, кодирующее положение разреза во входной последовательности (рассматриваемой как последовательность байт). На втором этапе решения задачи поиска SCS мы сводим её к задаче коммивояжёра, при этом каждая подстрока, полученная первым алгоритмом, имеет номер в соответствии со своим положением в файле. Матрица расстояний для вершин i и j , соответствующих подстрокам s_i и s_j , содержит длину префикса s_j , являющегося суффиксом s_i . Для ЗК ген — это номер одной из вершин графа маршрутов, а порядок генов определяет цепь, которая соединяет эти вершины.

Гены одной хромосомы мутируют с некоторой вероятностью независимо друг от друга. В случае генерации разрезов гены непосредственно изменяют своё значение. В случае ЗК гены меняют своё положение в хромосоме, что приводит к изменению пути.

При наследовании гены переходят в дочерние хромосомы целиком, кроссинговер осуществляется по границам генов. Поскольку цель нашей работы — поиск приближённого решения задачи построения кратчайшей суперстроки для произвольных данных, далее мы сосредоточим внимание на разработанном нами алгоритме.

Что касается решения ЗК, то нет единого мнения, какой из эвристических алгоритмов более эффективен для её решения, в большей части это зависит от качества реализации того или иного варианта. Изначально предполагалось, что мы будем пробовать совместить исходный алгоритм разбиения с различными вариантами алгоритма для решения ЗК.

2.2. Описание ГО для алгоритма разбиения файла на подстроки

Для задач с целочисленным кодированием генов предусмотрены специальные алгоритмы мутации и кроссинговера, немного отличающиеся от классических. Предполагаем, что нашему ГА соответствуют параметры, определяющие максимальное значение гена, которое по крайней мере определяется имеющимся входным файлом (обозначим это число *chromosomeLenMax*), и максимальную длину хромосом (обозначим *geneAmountMax*). Предлагаются следующие мутации:

- 1) случайное изменение одного из генов, реализуемое как генерация случайного целого числа n , имеющего равномерное распределение и не превышающего *chromosomeLenMax*, которое не совпадало бы с уже имеющимися в списке генов хромосомы;
- 2) прибавление или вычитание 1 (переход на соседнюю позицию) из текущего значения гена по модулю *chromosomeLenMax* и проверка генов на совпадение;
- 3) прибавление к одному из генов случайного числа n , имеющего гауссово распределение, по модулю *chromosomeLenMax* и проверка генов на совпадение.

Далее на базе этих функций были построены более агрессивные функции мутации, меняющие половину или произвольное число генов хромосомы, подвергавшейся мутации.

Число хромосом, которые подвергнутся мутации, устанавливается параметрами запуска алгоритма, и поскольку наш алгоритм построен на основе ПГА, то это значение не должно обычно превышать 1–5 % от общего числа хромосом в популяции [1].

Пример оператора мутации при использовании случайных чисел, имеющих гауссово распределение:

```
void GaussianMutationN(Chromosome chromosome, int n)
{
    GaussianRandom gaussianRandom =
        new GaussianRandom();
    if (n > chromosome.genes.Count())
        n = chromosome.genes.Count();
    // gen positions, where we'll change value
    List<int> positions = new List<int>();
    int position;
```

```

    position = random.Next(0, chromosome.genes.Count());
    positions.Add(position);
    for (int i = 0; i < n-1; i++)
    {
        position = random.Next(0, chromosome.genes.Count());
        while (positions.Contains(position))
        position = random.Next(0, chromosome.genes.Count());
        positions.Add(position);
    }
    Gene newAmount; // create gene
    for (int i = 0; i < n; i++)
    {
        // into gene construction value will be taken in
        newAmount = new Gene(chromosome.genes[position].iAmount
        + Convert.ToInt32(gaussianRandom.NextGaussian(Gene.maxAmount/2,
        Gene.maxAmount / 6)));
        //there must not be any duplicate genes in chromosome
        while (chromosome.genes.Contains(newAmount))
        newAmount = new Gene(chromosome.genes[position].iAmount
        + Convert.ToInt32(gaussianRandom.
        NextGaussian(Gene.maxAmount / 2,
        Gene.maxAmount / 6)));
        chromosome.genes[position].iAmount = newAmount.iAmount;
    }
}
//gaussian mutation for some genes
void GaussianMutationRandom(Chromosome chromosome)
{
    int n = random.Next(0, maxLength);
    GaussianMutationN(chromosome, n);
}

```

Дополнительно для нашего случая подходят мутации вставки и удаления генов. При этом необходимо контролировать максимальное и минимальное число генов в хромосоме, а при вставке, как указывалось выше, проверять наличие повторных генов.

Для нашего случая подходят такие операторы кроссинговера, которые, возможно, изменяли бы число генов в хромосоме, однако не давали бы одному и тому же гену появляться повторно. Были реализованы следующие операторы: одноточечный, многоточечный, универсальный кроссинговер, а также кроссинговер типа Cut-and-Split. Первые три из указанных операторов сохраняют первоначальную длину хромосом, последний позволяет склеивать куски хромосом произвольной длины. При выполнении кроссинговера также необходимо проверять хромосомы на корректность. Поскольку оператор Cut-and-Split менее известен и не относится к классическим, приведём его код:

```

List<Chromosome>
CutAndSplitCrossoverWithoutRef(Chromosome parent1,
                               Chromosome parent2)
{
    List<Chromosome> childs = new List<Chromosome>();

```

```

        childs.Add(new Chromosome());
        childs.Add(new Chromosome());
        int point1, point2; // crrossigover points
        point1 = random.Next(0, parent1.genes.Count);
        point2 = random.Next(0, parent2.genes.Count);
// it doesn't fit, if length is bigger then maximum or is 0
while (((point1 + parent2.genes.Count - point2) > maxLength)
    || ((point1 + parent2.genes.Count - point2) == 0)
    || ((point2 + parent1.genes.Count - point1) > maxLength)
    || ((point2 + parent1.genes.Count - point1) ==0))
    {
        point1 = random.Next(0, parent1.genes.Count);
        point2 = random.Next(0, parent2.genes.Count);
    }
    for (int i = 0; i < point1; i++) //обмен частями
    {
        if (!(childs[0].genes.Contains(parent1.genes[i])))
            childs[0].genes.Add(parent1.genes[i]);
    }
    for (int i = 0; i < point2; i++)
    {
        if (!(childs[1].genes.Contains(parent2.genes[i])))
            childs[1].genes.Add(parent2.genes[i]);
    }
    for (int i = point2; i < parent2.genes.Count; i++)
    {
        if (!(childs[0].genes.Contains(parent2.genes[i])))
            childs[0].genes.Add(parent2.genes[i]);
    }
    for (int i = point1; i < parent1.genes.Count; i++)
    {
        if (!(childs[1].genes.Contains(parent1.genes[i])))
            childs[1].genes.Add(parent1.genes[i]);
    }
    return childs;
}

```

Для выбора родительской пары используются методы:

- панмиксия — скрещивание случайных хромосом;
- инбридинг — близкородственное скрещивание, под термином будем понимать хромосомы, имеющие схожее значение функции пригодности;
- аутбридинг — скрещивание наиболее непохожих хромосом, под этим мы будем понимать хромосомы с максимально отличающимся значением функции пригодности;
- селекционное или элитарное скрещивание — скрещивание между собой лучших представителей.

Для имбридинга и аутбридинга сначала сортируем популяцию по значению функции пригодности, затем выполняем скрещивание в соответствии с полученным порядком. Пример аутбридинга:

```

    void PairSelectionOutbridging(delegate_Chrossover Crossover)
    {
// outbreeding, first is taken randomly, second—the most close
List<IRouletteWheelSector>
collection = new List<IRouletteWheelSector>();
    foreach (Chromosome chromosome in currentGeneration)
    { collection.Add(chromosome);
    }
    RouletteWheel rouletteWheel = new RouletteWheel(collection);
    int first_parent_num;
    first_parent_num = rouletteWheel.Spin();
    parents.Add(currentGeneration[first_parent_num]);
    int fitness_difference; // difference between fitnesses
    int second_parent_num; // parent's number
    second_parent_num = first_parent_num;
    fitness_difference = 0;

        for (int i = 0; i < currentGeneration.Count; i++)
        {
if ((i != first_parent_num)
    && (Abs(currentGeneration[i].fitness_
        - currentGeneration[first_parent_num].fitness_)
        > fitness_difference))
    {
        fitness_difference = Abs(currentGeneration[i].fitness_
        - currentGeneration[first_parent_num].fitness_);
        second_parent_num = i;
    }
        }
    parents.Add(currentGeneration[second_parent_num]);
    childs = Crossover(parents[0], parents[1]);
    }

```

В качестве оператора отбора нам подойдёт любой из известных, например процентный элитарный, рулеточный или ранговый, а также случайный на основе равной вероятности хромосом выжить или умереть.

Чтобы избежать потери перспективных хромосом, можно использовать процентный элитарный отбор.

Необходимо обратить внимание на то, что не любая произвольная комбинация ГО позволит получить хороший ГА. В некоторых случаях, например при сочетании случайного формирования родительских пар и случайного отбора, нет никакой гарантии роста значения функции пригодности и схождения алгоритма. Более того, мы можем утратить хорошие решения, поэтому требуется дополнительно проверить работоспособность алгоритма.

Алгоритм предполагает, что можно использовать различные функции пригодности, которые соответствовали бы определённому шаблону. Были реализованы функция пригодности на основе решения задачи о назначении и функция пригодности на основе суммирования элементов матрицы, описывающей пересечения строк полученного разбиения, что даёт более грубую, но и более быструю оценку полученного разбиения.

2.3. Формирование начальной популяции для алгоритма разбиения файла на подстроки

Если нам известно что-то о структуре входных данных, то мы можем предположить, как именно можно будет эффективно разбить файл для дальнейшего сжатия, в частности какой длины должны быть подстроки для сжатия.

Однако в общем случае мы не можем заранее сделать однозначный вывод о том, в каких позициях лучше разбить файл, чтобы получить наиболее короткую суперстроку. Мы решили генерировать начальную популяцию полностью случайным образом.

Также алгоритму необходимо передать параметры, задающие максимальное значение гена, которое по крайней мере определяется имеющимся входным файлом (обозначим это число *chromosomeLenMax*), и максимальную длину хромосом (обозначим *geneAmountMax*). Начальный размер популяции также будет в нашем случае входным параметром алгоритма, обозначим его *populationSize*.

Корректный выбор параметров алгоритма представляет собой отдельную и достаточно сложную задачу. При этом в простейшем случае предъявляется требование о том, что любое допустимое решение должно быть получено с ненулевой вероятностью в ходе работы ГА.

Опишем непосредственно алгоритм генерации начальной популяции. Для каждой новой хромосомы выполняются следующие два действия:

- 1) генерируется длина хромосомы $n \leq chromosomeLenMax$; длины хромосом выбираются в соответствии с равномерным распределением;
- 2) генерируется n генов в виде целых чисел, характеризующих позицию разрезов исходного файла; эти числа также выбираются с помощью равномерного распределения.

В зависимости от алгоритма размер популяции в дальнейшем может меняться или сохраняться. Также текущее поколение может полностью замещаться новым, полностью сохраняться или частично сохраняться.

3. Описание двух реализаций ГА с ДРВ

На основе работ [2] и [4] нами предлагаются две собственные интерпретации ГА с ДРВ.

3.1. ГА с фиксированием использованного набора ГО

В ходе выполнения ГА на каждом шаге в поколение будет добавляться ровно одна новая хромосома, которая заменит худшую хромосому из старого поколения. Таким образом, размер поколения всегда будет равен *populationSize*. Для каждой хромосомы будет фиксироваться, каким набором генетических операторов она получена. Вероятность применения того или иного генетического оператора будет зависеть от числа особей в текущей популяции, полученных с помощью этого оператора. Количество итераций *numberOfIterations* алгоритма будет изменяемым параметром, сообщаемым при старте алгоритма. Мутация каждого потомка происходит с вероятностью *mutationProbability*, которая также является задаваемым параметром.

Для каждого поколения ГА конкретный набор ГО формируется по алгоритму рулетки на основе вычисленных вероятностных характеристик каждого оператора. При инициализации алгоритма для каждого типа ГО формируется группа

функций, реализующих его. Для каждой функции в своей группе задаётся вероятность её применения. Распределение вероятностей первоначально равномерное. В каждой хромосоме, полученной в данной итерации, фиксируется тройка операторов данной итерации. Прежде чем накопить статистику об использовании ГО, мы должны проделать некоторое количество итераций алгоритма, обозначим это число за *numberOfIterationsFirstReinitialize*. Далее распределение вероятности операторов будет корректироваться каждые *numberOfIterationsReinitialize* шагов. Вероятность использования каждого ГО вычисляется как отношение числа живых хромосом в популяции, полученной с помощью данного ГО, к общему числу хромосом, для которых определены ГО, с помощью которых они получены. Хромосомы, оставшиеся от инициализации популяции, исключаются из рассмотрения.

Сами ГО выбираются следующим образом :

1. При инициализации алгоритма для каждого типа операторов — S , C , M — формируется группа функций, реализующих различные виды группирования в пары, кроссинговера и мутации соответственно. Далее рассматриваем *ParentPairs*, *Chrossover*, *Mutation* как коллекцию соответствующих функций.
2. Для каждого оператора в своей группе вычислена его вероятность применения. В начальный момент времени вероятности для всех операторов в группе равны.
3. На каждом шаге ГА конкретный набор из трёх операторов формируется с помощью алгоритма рулетки. В каждой хромосоме, полученной в данной итерации, фиксируется тройка операторов.
4. Прежде чем накопить статистику об использовании ГО, мы должны проделать некоторое количество итераций алгоритма, обозначим это число за *numberOfIterationsAshkyeReinitialize*. Далее частоты будут пересчитываться каждые *numberOfIterationsReinitialize* шагов.
5. Мы установим вероятность использования каждого ГО как отношение числа живых хромосом в популяции, полученной с помощью данного ГО, к общему числу хромосом, для которых определены ГО, с помощью которых они получены, то есть на число хромосом, полученных в ходе выполнения итераций ГА, а не сохранившихся из начальной популяции.

Возможны различные модификации алгоритма. Например, популяция может иметь нефиксированный размер. В этом случае к популяции можно добавлять всех полученных потомков либо потомков, прошедших оценку функцией пригодности (сама функция пригодности может выбираться аналогично другим генетическим операторам). Минус такого подхода — размер популяции наиболее существенно скажется на времени выполнения. В популяцию можно попытаться добавить всех полученных потомков, пригодность которых больше, чем пригодность худших особей в популяции. Кроме того, можно определить критерий сходимости популяции следующим образом: если в течение N итераций полученные потомки имеют худшую (или не лучшую) приспособленность, чем особи в популяции, то можно считать, что популяция сошлась. В качестве ответа можно вернуть лучшую особь в текущей популяции. Плюс такого подхода — на каждой итерации нам надо пересчитать функцию пригодности только для полученных потомков *children*. Также если популяция отсортирована, то новые элементы легко добавить. Минусы по сравнению со следующим подходом — в каждой итерации генерируется меньше элементов, и, соответственно, необходимо увеличивать число итераций.



ГА с ДРВ с оценкой ГО по среднему значению пригодности

3.2. ГА с ДРВ с оценкой ГО по среднему значению пригодности

В ходе выполнения ГА на каждой итерации генерируется некоторое количество новых хромосом. Старое поколение будет полностью заменено новым поколением, сформированным из числа лучших хромосом, полученных на этой итерации. Для регулирования численности будет применяться оператор рекомбинации. При этом размер полученного поколения всегда будет *populationSize*. Если на данном этапе было получено меньше новых хромосом, чем *populationSize*, то лучшие хромосомы из детей войдут в новое поколение несколько раз. Схема алгоритма представлена на рисунке. Определение числа итераций и мутация потомков происходят аналогично описанному в разд. 3.1 алгоритму.

Аналогично предыдущему алгоритму при инициализации для каждого типа операторов формируется группа функций, реализующих соответствующие ГО: отбор родительских хромосом, выбор родительской пары, кроссинговер, мутация, рекомбинация. В начальный момент времени вероятности для всех операторов в группе задаются равными. На каждом шаге ГА конкретный набор из пяти операторов формируется с помощью алгоритма рулетки.

С каждой из групп функций связывается два счётчика, один из которых показывает число хромосом, задействованных для данного оператора в текущем поко-

лении, а второй — число полученных прогрессивных хромосом, то есть хромосом, для которых значение функции пригодности выше, чем среднее значение функции пригодности в родительской популяции. После выполнения каждого оператора фиксируем количество прогрессивных хромосом в популяции и общее число хромосом, сгенерированных алгоритмом. На основании этих частот пересчитываем вероятность выбора соответствующего генетического оператора как отношение количества полученных этим оператором прогрессивных хромосом к общему количеству прогрессивных хромосом.

При реализации алгоритма в таком виде (даже без оценки пригодности и пересчёта вероятности использования ГА) в проведённых нами тестах наблюдается существенный рост времени работы алгоритма в сравнении с аналогичными детерминированными аналогами. Наиболее трудозатратными оказались операции по пересчёту значений функции пригодности для всей популяции и сортировка популяции по значению функции пригодности. Вследствие этого значительное время занимает вычисление оператора редукции в соответствии со значением функции пригодности. Этот оператор требует значительно больше времени на выполнение в сравнении с другими операторами. Однако это единственный оператор редукции, который гарантированно позволяет увеличивать значение функции пригодности в новой популяции. Без этого оператора некоторые варианты сочетания селекции, кроссинговера и мутации будут носить абсолютно случайный характер. Таким образом, данный алгоритм требует оптимизации по времени, а также очень сложен для статистического анализа, так как многие величины в ходе выполнения алгоритма будут случайными, при этом разные генетические операторы будут менять эти величины по-разному (размер пула родителей и число потомков, в частности).

Список литературы

1. **Гладков, Л. А.** Генетические алгоритмы / Л. А. Гладков, В. В. Курейчик, В. М. Курейчик. — М. : Физматлит, 2006. — 320 с.
2. **Бюргер, Ю. А.** Мягкие вычисления [Электронный ресурс] / Ю. А. Бюргер. — URL: http://www.getinfo.ru/article28_3.html (дата обращения: 20.03.2016).
3. **Петров, Ю. Ю.** Применение генетического алгоритма с регуляцией вероятностей генетических операторов при решении задачи упаковки в контейнеры / Ю. Ю. Петров // Сб. науч. тр. Сев.-Кавказ. гос. тех. ун-та. Сер. естественнауч. — 2006. — № 2.
4. **Мясников, А. С.** Островной генетический алгоритм с динамическим распределением вероятностей выбора генетических операторов [Электронный ресурс] / А. С. Мясников. — URL: <http://technomag.edu.ru/doc/136503.html> (дата обращения: 27.05.2012).
5. **Gallant, J.** On finding minimal length superstrings / J. Gallant, D. Maier, J. A. Storer // J. of Computer and System Sciences. — 1980. — Vol. 20, no. 1. — P. 50–58.
6. Linear approximation of shortest superstring / A. Blum [et al.] // J. of the ACM. — 1994. — Vol. 4, iss. 4. — P. 630–647.
7. **Kohn, S.** A generating function approach to the traveling salesman problem / S. Kohn, A. Gottlieb, M. Kohn // Proceedings of the 1977 annual conference ACM '77. — New York, 1977. — P. 294–300.
8. An algorithm for the traveling salesman problem / J. D. C. Little [et al.] // Operations Research. — 1963. — Vol. 11, iss. 6. — P. 972–989.
9. **Sweedyk, Z.** A 2.5-approximation algorithm for shortest superstring / Z. Sweedyk // SIAM J. of Computing. — 1999. — Vol. 29, no. 3. — P. 954–986.

10. **Liu, X.** Algorithms for the shortest common superstring problem / X. Liu, O. Sykora // Parallel Numerics '05: Theory and Application / ed. by M. Vajteršic [et al.]. — Ljubljana ; Salzburg : Jožef Stefan Inst. : Univ. of Salzburg, 2005. — P. 97–107.
11. **Zaritsky, A.** Coevolving solution to the shortest common superstring problem / A. Zaritsky, M. Sipper // BioSystems. — 2004. — Vol. 76, no. 1. — P. 209–216.

Поступила в редакцию 28.04.2016

После переработки 07.06.2016

Сведения об авторе

Кашкарева Юлия Олеговна, аспирантка математического факультета, Челябинский государственный университет, Челябинск, Россия; e-mail: kashkareva.julija@gmail.com.

Chelyabinsk Physical and Mathematical Journal. 2016. Vol. 1, iss. 2. P. 24–36.

GENETIC ALGORITHM WITH A DYNAMIC PROBABILITIES DISTRIBUTION OF THE SELECTION OF GENETIC OPERATORS FOR SOLVING OF PROBLEMS WITH INTEGER GENES CODING

Yu.O. Kashkareva

Chelyabinsk State University, Chelyabinsk, Russia

kashkareva.julija@gmail.com

Two genetic algorithms are designed and tested for constructing of the superstring for an arbitrary binary file. They are based on the adaptive choosing of genetic operators set.

Keywords: *superstring construction, genetic algorithm, travelling salesman problem.*

References

1. **Gladkov L.A., Kureychik V.V., Kureychik V.M.** *Geneticheskiye algoritmy* [Genetic algorithms]. Moscow, Fizmatlit Publ., 2006. 320 p. (In Russ.).
2. **Byurger Yu.A.** *Myagkiye vychisleniya* [Soft computing]. Available at: http://www.getinfo.ru/article28_3.html, accepted 20.03.2016. (In Russ.).
3. **Petrov Yu.Yu.** Primenenie geneticheskogo algoritma s regulyatsiyey veroyatnostey geneticheskikh operatorov pri reshenii zadachi upakovki v konteynery [Application of genetic algorithm to the regulation of the probability of genetic operators in solving of the bin packing problem]. *Sbornik nauchnykh trudov Severo-Kavkazskogo gosudarstvennogo tekhnicheskogo universiteta. Seriya Yestestvennonauchnaya* [Proceedings of North Caucasus State Technical University. Nature Sciences Series], 2006, no. 2. (In Russ.).
4. **Myasnikov A.S.** Ostrovnoy geneticheskiy algoritm s dinamicheskim raspredeleniyem veroyatnostey vybora geneticheskikh operatorov [Island algorithm with dynamic probabilities distribution of genetic operators choosing]. Available at: <http://technomag.edu.ru/doc/136503.html>, accepted 27.05.2012. (In Russ.).
5. **Gallant J., Maier D., Storer J.A.** On finding minimal length superstrings. *Journal of Computer and System Sciences*, 1980, vol. 20, no. 1, pp. 50–58.
6. **Blum A.** [et al.]. Linear approximation of shortest superstring. *Journal of the ACM*, 1994, vol. 41, iss. 4, pp. 630–647.
7. **Kohn S., Gottlieb A., Kohn M.** A generating function approach to the traveling salesman problem. *Proceedings of the 1977 annual conference ACM '77*. New York, 1977. Pp. 294–300.

8. **Little J.D.C.** [et al.]. An algorithm for the traveling salesman problem. *Operations Research*, 1963, vol. 11, iss. 6, pp. 972–989.
9. **Sweedyk Z.** A 2.5-approximation algorithm for shortest superstring. *SIAM Journal of Computing*, 1999, vol. 29, no. 3, pp. 954–986.
10. **Liu X., Sykora O.** Algorithms for the shortest common superstring problem. *Parallel Numerics '05: Theory and Application*, ed. by M. Vajteršic [et al.]. Ljubljana, Jožef Stefan Institute; Salzburg, University of Salzburg, 2005. Pp. 97–107.
11. **Zaritsky A., Sipper M.** Coevolving solution to the shortest common superstring problem. *BioSystems*, 2004, vol. 76, no. 1, pp. 209–216.

Accepted article received 28.04.2016

Corrections received 07.06.2016