Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики

Факультет информационных технологий и программирования
Кафедра компьютерных технологий

Маркина Маргарита Анатольевна

Разработка гибридного алгоритма недоминирующей сортировки

Научный руководитель: к.т.н. доцент кафедры КТ М. В. Буздалов

Санкт-Петербург
2016

Содержание

Введение

Множество известных и широко распространенных многокритериальных эволюционных алгоритмов используют процедуру недоминирующей сортировки, или процедуру определения множества недоминирующих решений, которая может быть сведена к недоминирующей сортировке. Примерами таких алгоритмов могут послужить NSGA-II [1], PESA [2], PESA-II [3], SPEA2 [4], PAES [5], PDE [6] и многие другие алгоритмы. Вычислительная сложность одной итерации этих алгоритмов часто определяется сложностью процедуры недоминирующей сортировки, следовательно, снижение сложности последней делает такие многокритериальные эволюционные алгоритмы значительно быстрее.

Существуют разные алгоритмы недоминирующей сортировки, но эффективность их работы очень сильно отличается в зависимости от данных. Этим можно воспользоваться и совместить идеи разных алгоритмов в одном, чтобы получить новый алгоритм, сочетающий в себе преимущества существующих, избавившись при этом от их недостатков.

Цель данной работы – сделать гибридный алгоритм недоминирующей сортировки, который будет использовать наиболее подходящий алгоритм или переключаться между алгоритмами в ходе своей работы.

В Главе 1 данной работы представлен общий обзор работы. В разделе 1.1 подробно рассмотрены определение недоминирующей сортировки и необходимые для этого понятия, а также представлены примеры применения недоминирующей сортировки, подтверждающие актуальность данной работы. В разделе 1.2 произведен обзор имеющихся результатов и подробно описаны лучшие из них. В разделе 1.3 описаны недостатки существующих алгоритмов. В разделе 1.4 сформулирована постановка задачи.

В Главе 2 представлены теоретические исследования по гибридизации алгоритмов недоминирующей сортировки. В разделе 2.1 произведен

анализ существующих алгоритмов и их сравнение. В разделе 2.2 показаны основные проблемы, возникающие при гибридизации алгоритмов, а также предложены пути их решения.

В Главе 3 представлены практические исследования и их результаты. В разделе 3.1 представлена общая архитектура программы и использованные оптимизации. В разделе 3.2 приведены данные экспериментов по сравнению скорости работы гибридного алгоритма относительно старых. В разделе 3.3 показано улучшение производительности практической задачи, использующей разработанный алгоритм для недоминирующей сортировки.

В заключении подведены итоги работы, а также сказано, какие могут быть дальнейшие пути развития гибридных алгоритмов недоминирующей сортировки.

Глава 1. Обзор работы

В этой главе представлен общий обзор работы: уточнены цели и объяснены термины и понятия, присутствующие в решении задачи. Также произведен обзор имеющихся результатов и сформулирована постановка задачи.

1.1. Недоминирующая сортировка

В данном разделе представлено определение недоминирующей сортировки и необходимые для ее понимания понятия. Также рассмотрены случаи применения недоминирующей сортировки, которые обосновывают актуальность нового ускоренного алгоритма недоминирующей сортировки.

1.1.1. Определение

Недоминирующая сортировка – это процедура, которая ранжирует множество точек в многомерном пространсве R^n. Если описывать нефор-мально, то ее задача определить, какие точки ¨учше¨ругих. При этом до-пускается, что две точки могут быть одинаково ¨орошими:

Для того, чтобы сформулировать определение недоминирующей сортировки, сначала надо определить, какие точки мы считаем ¨уч-ше¨ругих. Для этого введем определение доминирования одной точки дру-гой.

Определение. В M-мерном пространстве, точка $A = (a_1, ..., a_M)$ до-минирует точку $B = (b_1, ..., b_M)$ тогда и только тогда, когда для всех $1 \leq i \leq M$ выполняется неравенство $a_i \leq b_i$, и существует такое j, что $a_j < b_j$.

¨учшими¨ данном контексте будут считаться точки, которые не до-минируются ни одной другой точкой или, другми словами, лежащие на Парето-фронте. Однако часто бывают не только точки с парето-фронта,

но и другие ¨орошие¨очки. Таким образом мы приходим к определению процедуры недоминирующей сортировки.

Определение. Недоминирующая сортировка множества точек S в M-мерном пространстве — это процедура, которая назначает всем точкам из S ранг. Все точки, которые не доминируются ни одной точкой из S имеют ранг ноль. Точка имеет ранг $i+1$, если максимальный ранг среди доминирующих её точек равен i.

1.1.2. Применение и актуальность

Самый яркий пример применения процедуры недоминирующей сортировки – алгоритмы многокритериальной оптимизации, особенно эволюционные алгоритмы. Последние на каждой итерации генеририруют множество потенциальных решений и оценивают каждое решение по всем критериям. Если критериев M, то получается набор из N M-мерных векторов, где N – число потенциальных решений. И эволюционому алгоритму на каждой итерации надо отобрать лучшие решения, для чего и требуется недоминирующая сортировка.

Если каждый критерий для всех каждого потенциального решения считается достаточно долго, то время выполнения недоминирующей сортировки становится неважным, так как асимптотика каждой итерации алгоритма зависит в основном от асимптотики времени подсчета критериев. Однако гораздо чаще встречаются задачи, в которых подсчет каждого критерия занимает время значительно меньшее, чем время, еобходимое для недоминирующей сортировки. Именно в таких случаях ускорение алгоритмов недоминирующей сортировки ускорит время выполнения итерации алгоритма, а следовательно и время выполнения всего алгоритма.

В настоящее время существует много алгоритмов недоминирующей сортировки. Но каждый из них имеет свои слабые стороны. Это означает, что существует возможность сделать алгоритм, который мог бы заранее предсказывать, время работы какого алгоритма на данном наборе точек

будет меньше, и выбирать оптимальный. Более того есть возможность совместить идеи разных алгоритмов в одном, который всегда будет работать не хуже существующих. Такие возможности делают проблему ускорения алгоритмов недоминирующей сортировки еще более актуальной.

1.2. Анализ существующих алгоритмов

В данном разделе будут рассмотрена история развития алгоритмов недоминирующей сортировки. Особое внимание будет уделено самым эффективным алгоритмам, которые применяются для гибридизации в данной работе. Они будут рассмотрены наиболее подробно.

1.2.1. Наивные алгоритмы

Опишем самый наивный алгоритм недоминирующей сортировки. Он перебирает все пары точек и сравненивает их по всем критериям. После этого он присваивает нулевой ранг тем из них, которые не доминируются ни одной другой точкой и отбрасывает их из множества. Данная процедура повторяется, пока в множестве остаются точки. Причем на каждом новом шаге присваивается новое значение ранга, на единицу больше, чем на предыдущем шаге. Рассмотрим время работы данного наивного алгоритма. Пусть N — это число точек, а M — размерность пространства. Тогда сравнение всех пар точек по M критериям займет $O(MN^2)$, а всего шагов алгоритма будет не больше, чем максимальное число рангов – N. Таким образом, время работы данного алгоритма не превышает $O(MN^3)$, причем эта оценка достигается в худшем случае при максимальном числе рангов в сортируемом множестве.

В работе Кунга и др. [7] предлагается алгоритм определения множества недоминируемых точек, при этом его вычислительная сложность составляет $O(Nlog^{M-1}N)$. Этот алгоритм возможно использовать для выполнения недоминирующей сортировки аналагично вышеописанному алгоритму. Сначала в множестве S алгоритм Кунга находит множество точек

с рангом 0. Затем алгоритм Кунга запускается на оставшемся множестве точек, и получившемуся множеству точек присваивается ранг 1. Процесс выполняется до тех пор, пока имеются точки, которым не присвоен ранг. Описанная процедура в худшем случае выполняется за $O(N^2 log^{M-1} N)$, если максимальный ранг точки равен $O(N)$.

Также существует много других алгоритмов, асиптотика которых равна $O(MN^2)$, например, алгоритм ENS Жанга и др. [**Zhang**].

1.2.2. Алгоритмы «Разделяй и властвуй»

Йенсен [8] впервые предложил алгоритм недоминирующей сортировки с вычислительной сложностью $O(Nlog^{M-1} N)$. Однако, как корректность, так и оценка сложности алгоритма доказывалась в предположении, что никакие две точки не имеют совпадающие значения ни в какой размерности. Однако довольно часто алгоритмы оптимизации работают с дискретными критериями, поэтому совпадение разных решений по одному критерию может быть довольно частым событием. Устранить указанный недостаток оказалось достаточно трудной задачей — первой успешной попыткой сделать это, насколько известно исполнителю данной НИР, является работа Фортена и др. [9]. Исправленный (или, согласно работе, «обобщенный») алгоритм корректно работает во всех случаях, и во многих случаях его время работы составляет $O(Nlog^{M-1} N)$, но единственная оценка времени работы для худшего случая, доказанная в работе [8], равна $O(N^2 M)$. Наконец, в работе Буздалова и др. [10] предложены модификации алгоритма из работы [8], которые позволили доказать в худшем случае также и оценку $O(Nlog^{M-1} N)$, не нарушая корректности работы алгоритма.

Опишем подробнее алгоритм Буздалова и др., так как он будет использоваться в гибридном алгоритме, разрабатываемом в данной работе. Основная идея алгоритма – принцип “разделяй и властвуй”, основанный на разбиении исходного множества на несколько меньших множеств и реше-

нии задачи на этих множествах. Алгоритм на каждом шаге находит медиану множества по последнему критерию и делит его на три подмножества элементов, меньших медианы по последнему критерию, больших и равных ей. Далее алгоритм рекурсивно запускается на каждом подмножестве с некоторыми промежуточными вычислениями.

Рассмотрим подробнее процедуры, использующиеся в алгоритме Буздалова. Основными из них являются процедуры $NDHelperA$, $NDHelperB$ и $SplitBy$. Первая как раз и представляет собой основной алгоритм на множестве S, которое дается ему на вход. Однако она может также присвоить ранги точкам так, чтобы они не стали меньше, чем ранги, которые эти точки имели до исполнения процедуры. Также данная процедура сравнивает точки только по первым k критериям. Эти дополнения необходимы для возможности корректного рекурсивного вызова данной процедуры. Алгоритм Буздалова, по сути, заключается в том, что расставляет всем точкам множества S ранг ноль, а затем запускает процедуру $NDHelperA$ с аргументами S и M.

Процедура $NDHelperA$ не работает с размерностями меньше 2, так как для них есть более эффективные алгоритмы, которые она и запускает при необходимости.

Псевдокод $NDHelperA$ представлен на рисунке 1.1.

Следующая процедура $NDHelperB$ запускается между рекурсивными запусками $NDHelperA$ на трех подмножествах. Задача этой процедуры – расставить минимально возможные ранги точек для подмножества, на котором сейчас запустится $NDHelperA$. Простая имплементация данной процедуры могла бы перебрать все пары точек из множества точек с уже проставленными рангами и точек, на которых сейчас запустится $NDHelperA$, и проставить каждой точке из второго множества ранг, на единицу большие, чем максимальный ранг точки с уже проставленным рангом, которая ее доминирует. Однако это работало бы квадратичное время по размеру подмножеств, поэтому данная процедура также использует

```
1: procedure NDHelperA(S, k)
2:     if |S| < 2 then return
3:     else if |S| = 2 then
4:         {s^(1), s^(2)} ← S
5:         if s^(1)_{1:k} ≺ s^(2)_{1:k} then
6:             RANK(s^(2)) ← max{RANK(s^(2)), RANK(s^(1)) + 1}
7:         end if
8:     else if k = 2 then
9:         SweepA(S)
10:    else if |{s_k | s ∈ S}| = 1 then
11:        NDHelperA(S, k − 1)
12:    else
13:        L, M, H ← SplitBy(S, median{s_k | s ∈ S}, k)
14:        NDHelperA(L, k)
15:        NDHelperB(L, M, k − 1)
16:        NDHelperA(M, k − 1)
17:        NDHelperB(L ∪ M, H, k − 1)
18:        NDHelperA(H, k)
19:    end if
20: end procedure
```

Рис. 1.1: Процедура NDHelperA. Она присваивает ранги точкам из S по первым k рангам.

принцим "разделяй и властвуй" и при расставлении минимальных рангов также разбивает множества на более мелкие и запускается на них рекурсивно.

Более подробно процедура $NDHelperB$ описана в псевдокоде на рисунке 1.2.

Последняя процедура, которую стоит упомянуть для описания алгоритма Буздалова и др. – процедура разбиения множеств $SplitBy$. Она работает за линейное время по размеру множества, которое она разбивает и делит его на три множества: точки, большие m по критерию k, равные m и меньшие m. В каждом получившемся подмножестве сохраняется порядок точек, который был в оригинальном множестве.

Процедура $SplitBy$ описана в листинге на рисунке 1.3.

1.2.3. Алгоритм Роя и др

Большой интерес представляет алгоритм Роя *Best Order Sort* (*BOS*) [**Roy**], который в отличии вышеупомянутых не использует метод разделяй и властвуй. Его вычислительная сложность $O(MNlogM+MN^2)$. В лучшем случае алгоритм работает за $O(MNlogM)$,

```
1: procedure NDHelperB(L, H, k)
2:     if L = {} or H = {} then return
3:     else if |L| = 1 or |H| = 1 then
4:         for all h ∈ H, l ∈ L do
5:             if l_{1:k} ⪯ h_{1:k} then
6:                 rank(h) ← max{rank(h), rank(l) + 1}
7:             end if
8:         end for
9:     else if k = 2 then
10:        SweepB(L, H)
11:    else if max{l_k | l ∈ L} ≤ min{h_k | h ∈ H} then
12:        NDHelperB(L, H, k − 1)
13:    else if min{l_k | l ∈ L} ≤ max{h_k | h ∈ H} then
14:        m ← median{s_k | s ∈ L ∪ H}
15:        L_1, M_1, H_1 ← SplitBy(L, m, k)
16:        L_2, M_2, H_2 ← SplitBy(H, m, k)
17:        NDHelperB(L_1, L_2, k)
18:        NDHelperB(L_1, M_2, k − 1)
19:        NDHelperB(M_1, M_2, k − 1)
20:        NDHelperB(L_1 ∪ M_1, H_2, k − 1)
21:        NDHelperB(H_1, H_2, k)
22:    end if
23: end procedure
```

Рис. 1.2: Процедура NDHelperB. Она подгоняет ранги точек из H используя первые k критериев, сравнивая их с точками из L.

```
1: procedure SplitBy(S, m, k)
2:     L ← {s ∈ S | s_k < m}
3:     M ← {s ∈ S | s_k = m}
4:     H ← {s ∈ S | s_k > m}
5:     return L, M, H
6: end procedure
```

Рис. 1.3: Процедура разбиения. В каждом подмножестве спользуется такой же порядок точек, как и до разбиения

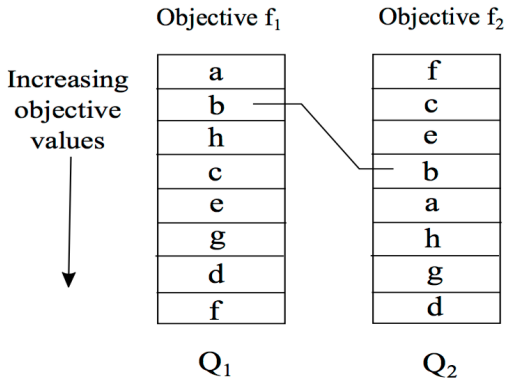

Рис. 1.4: На рисунке представлены отсортированные списки по критерию f1 и f2. Точка b будет сравниваться только с точкой a и в последствии ее ранг не будет меняться.

что лучше алгоритма предложенного Буздаловым и др. Однако в худшем случае его асимптотика совсем другая - $O(MN^2)$. Авторами алгоритма не было проведено более точных исследований по его времени работы.

Данный алгоритм создает отсортированный списки точек соответствующие каждому критерию (см. рис. 1.4). Если у точек критерии совпадают, используется порядок, основанный на предыдущих критериях. Далее просматриваем точки начиная с первых в этих списках, переходя от списка к списку, потом переходим ко вторым точкам в этих списках. Ранг назначаются точкам при первой встречи, если точка уже имеет ранг, она пропускается.

Более детальное описание алгоритма не так существенно, так как алгоритм BOS встраивается в алгоритм Fast, и его внутреннее устройство не очень интересно.

1.3. Недостатки существующих алгоритмов

Все описанные выше алгоритмы имеют разные асимптотики времени работы.

Квадратичные алгоритмы и алгоритмы с большей асимптотикой работают медленно на больших данных, однако при небольшом размере входного множества способны завершать работу быстрее алгоритмов “разделяй и властвуй” из-за того, что те вынуждены делать много дополнительной

работы, которая оказывается лишней на таких небольших размерах входных данных.

Алгоритмы “разделяй и властвуй” имеют хорошую асимптотику. В случае алгоритма Буздалова, даже на самых плохих входных данных. Однако они начинают работать гораздо хуже, когда размерность задачи M перестает быть константой, потому что экспоненциально зависят от его значения. Стоит признать, что случай больших M довольно редко встречается на практике, однако не явялется невозможным.

Последний из рассмотренных алгоритмов, алгоритм Роя, имеет интересную идею и показал хорошие результаты на практике. Однако теоретическое время его работы исследованно крайне плохо. Поэтому нельзя заране утверждать, всегда ли он лучше существующих алгоритмов.

1.4. Постановка задачи

Задача данной работы состоит в разработке нового гибридного алгоритма недоминирующей сортировки и разивается на подзадачи:

- Выбрать наиболее подходящие для гибридизации алгоритмы.
- Основываясь на практических экспериментах на разных видах входных данных, оценить время обработки каждым выбранным алгоритмом.
- Выдвинуть предположение о том, как и в какой момент менять стратегию сортировки.
- Проверить предположение
- Сформулировать и реализовать гибридный алгоритм.

Глава 2. Теоретические исследования

В данной главе будут представлены основные результаты работы. Сначала будет рассмотрен основные кандидаты для гибридизации. Затем будет представлен анализ работы кандидатов на разных входных данных. Потом будет сделано некоторое предположение на основании экспериментов и на его основе сформулирован алгоритм гибридизации.

2.1. Анализ существующих алгоритмов

В качестве основного кандидата для гибридизации был выбран алгоритм “разделяй и властвуй”. Причин для этого две:

1. Данный алгоритм работает лучше большинства алгоритмов. Благодаря этому гибридный алгоритм тоже будет работать эффективно.
2. Данный алгоритм рекурсивно запускает себя в процессе своей работы. Это порождает точки возможного подключения других алгоритмов.

Вторым кандидатом стал алгоритм Роя по следующим причинам:

1. Судя по разобранным авторами этого алгоритма случаям, данный алгоритм в лучших случаях может работать крайне эффективно, лучше других известных алгоритмов.
2. Нет строгого доказательства времени работы данного алгоритма, только результаты экспериментальных запусков. Данная работа является отличной возможностью сравнить этот алгоритм с другими существующими.

Далее в данной работе для краткости алгоритм Буздалова будет называться “Fast”, потому что он считается алгоритмом быстрой недоминирующей сортировки, а алгоритм Роя – “BOS”, так как его автор называ-

Рис. 2.1: Относительная разность времени работы кандидатов для гибридизации при разных способах генерации входных данных: $CUBE$ – случайные точки в гиперкубе, FN – случайные точки, имеющие N фронтов

ет свой алгоритм лучшим алгоритмом недоменирующей сортировки (Best Order Sort).

Был проведен ряд экспериментов, сравнивающий времена работы данных алгоритмов. В данных экспериментах сравнивалось среднее время работы каждого алгоритма на данных, имеющих разные размера и размерности. Для визуального представления и анализа были нарисованы графики функции $\frac{T_{BOS}-T_{Fast}}{T_{MAX}}$, где T_{BOS} – время работы алгоритма BOS, T_{Fast} – время работы алгоритма Fast, а $T_{MAX} = \max(T_{BOS}, T_{Fast})$. Чем меньше значение этой функции, тем эффективнее работает алгоритм BOS. Если функция принимает значение ноль, то оба алгоритма работают одинаково эффективно.

Эксперименты проводились для разных N и M – числа и размерности точек соответственно. Также использовались разные способы генерации входных данных:

1. Случайные точки в гиперкубе.
2. Случайно сгенерированные точки, имеющие фиксированное число фронтов.

Для подсчета времени работы алгоритма на каждых данных проводилось несколько его запусков. Время засекалось с помощью Java-библиоетки Если суммарное время даже нескольких запусков было очень маленьким и имело малую точность, алгоритм запускался на этих значениях еще больше раз, пока не достигалась желаемая точность измерений.

Ниже на рисунке 2.1 приведены результаты зависимости относительной разницы во времени работы алгоритмов от размера водных данных для разных размерностей и способов генерации входных данных.

Результаты экспериментов получились очень сильно зашумленными, поэтому для того, чтобы лучше увидеть зависимость времени работы

Рис. 2.2: Зависимость левой и правой границы эффективности BOS от размерности задачи:$CUBE$ – случайные точки в гиперкубе, FN – случайные точки, имеющие N фронтов

алгоритмов от размера входных данных, был применен метод интерквартериальных интервалов. Он заключается в том, что в каждом небольшом промежутке аргумента (в нашем случае это число точек сортируемого множества N) удаляется половина точек, которые находятся ближе к крайним значениям. Применение данного метода дало увидеть более четкую картину происходящего.

Из данных, полученных в результате экспериментов, можно сделать некоторые выводы. Самым главным утвержднием является то, что алгоритм Fast работает быстрее, чем BOS при очень малых и очень больших размерах входных данных. Однако сучществует некоторый интервал значений N, при котором алгоритм BOS работает значительно лучше. Причем можно рассмотреть зависимость левой и правой границ в зависимости от размерности задачи и способа генерации даных. Графики этих зависимостей представлены на расунке 2.2.

Нижняя и верхняя границы почти всегда зависят только от размерности и слабо зависят от числа фронтов. Зависимость от числа фронтов проявляется только у верхней границы: на графике для случайных точек в гиперкубе виден большой скачок при небольших размерностях. Это объясняется тем, что с ростом размерности при одинаковом числе входных точек значительно падает число фронтов, но при малых размерностях оно может быть большим. Таким образом, было выдвинуто предположение, что нижняя граница зависит от размерности M по закону $l(d) = c_1 \cdot d \cdot \ln(d+1)$, а верхняя – $r(d) = c_2 \cdot d \cdot (\ln^{0.9}(d+1) - 1.5)$, где c_1 и c_2 – константы, зависящие от машины, на которой работает алгоритм, а также от ее загруженнсоти. В случае машины, на которой запускались эксперименты, $c_1 \approx 1$, а $c_2 \approx 150$.

2.2. Предлагаемая схема гибридизации

Как уже говорилось выше, мы можем воспользоваться тем, что алгоритм Fast может во время рекурсивного запуска себя запускать другой алгоритм недоминирующей сортировки, в нашем случае BOS. Однако он должен уметь быстро определять, какой алгоритм в данном случае лучше запустить. Если решение будет занимать много времени, то алгоритм будет работать дольше, чем лучший из гибридиируемых алгоритмов.

Один из способов быстро выбрать алгоритм – основываясь на размерах и размерности данных и на результатах экспериментов сказать, какой алгоритм на этих данных отработает быстрее, и выбрать его. Вероятно, это не самый эффективный способ выбора момента переключения. Может случиться, что если алгоритм Fast сделает еще несколько шагов вглубь, создав множества поменьше, то алгоритм BOS на них отработает значительно лучше, и время всего алгоритма в целом будет значительно меньше. Однако момент оптимального переключения неизвестен, а также очень трудно находим с помощью экспериментов из больших шумов во времени работа гибрида.

Таким образом, прелагается следующий гибридный алгоритм:

1. Запускаем алгоритм Fast.
2. Перед каждым запуском $NDHelperA$ проверяем, не лучше ли на данном размере данных и размерности работает BOS.
3. Запускаем лучший из двух алгоритм.

2.2.1. Проблемы

В описанном алгоритме существуют две основные проблемы, которые сформулированы в данном подразделе.

2.2.1.1. Момент переключения

Первой проблемой является то, что заранее не может быть известно, какой алгоритм лучше на конкретных данных. Не может быть дано даже теоретических оценок зависимости лучшего алгоритма от размера и размерности входных данных, так как они отсутствуют для алгоритма BOS. Однако благодаря тому, что по результатам экспериментов, для каждой размерности входных точек, алгоритм BOS работает лучше только на определенном интервале значений числа входных точек, момент переключения легко вычислять по тому, находится ли текущий размер сортируемого множества в данном интервале.

При этом до сих пор остается проблемой вычиление границ интервала, на котором эффективнее работает BOS. Решение данной проблемы описано в следующем подразделе.

2.2.1.2. Предпоставленные ранги

Второй проблемой является то, что алгоритм Fast перед рекурсивным запуском себя расставляет минимальные ранги для точек множества, на котором собирается запускаться. Алгоритм BOS не предусматривает возможности наличия у точек минимально возможных рангов, поэтому требуется модификация этого алгоритма.

2.2.2. Выбор момента переключения

Как уже было сказано, основная проблема выбора момента переключения заключается в нахождении границ интервала, на котором BOS работает лучше, чем алгоритм Fast. Эти границы имеют завсимость от размерности входных данных, а также от можности машины, на которой запускается алгоритм. Ниже описаны методы нахождения этих зависимостией.

2.2.2.1. Зависимость от данных

В результате экспериментов было выявлено, что левая и правая граница интервала размеров входных данных, на котором BOS работает эффективнее, чем Fast, зависит от размерности входных точек, а также иногда от числа различных рангов во входном множестве.

Левая граница интервала почти не зависит от числа рангов, а ее поведение завимиости от размерности задачи достаточно хорошо описывается формулой $1/5M\ln(M+1)$.

С правой границей все гораздо хуже: она ведет себя непредсказуемо даже для фиксированного числа рангов во входных данных. Для решения данной проблемы требуется научиться считать эту границу перед запусками алгоритма.

2.2.2.2. Выбор времени переключения

Было предложено решение проблемы зависимости границ от данных: перед запуском алгоритма сначала искать границы эффективного интервала алгоритма BOS на случайных данных. Для этого на размерностях не более M, где M – это максимально возможная размерность, которая понадобится в задаче, запускается процедура нахождения границ. Она состоит из трех этапов: поиск размера данных, на котором алгоритм BOS максимально эффективнее алгоритма Fast, затем поиск левой границы и, наконец, поиск правой границы.

Первый этап реализован с помощью троичного поиска. Как мы видим из результатов экспериментов, производная функции относительной разницы времени работы алгоритмов имеет всего лишь один ноль производной. Таким образом, применив троичный поиск на интервале $[0; 2 \cdot 10^4]$, мы сможем найти минимум этой функции. Правая граница интервала поиска выбрана так, чтобы по результатам экспериментов она всегда значительно превышала правую границу интервала эффективности BOS, что гарантирует нам успешное нахождение минимума.

Стоит заметить, что так как модуль производной функции сильно падает с ростом N. Также мы можем в достаточной мере пренебречь точностью занчения границ при достаточно больших значениях входных данных. Из этого можно сделать вывод, что троичный поиск можно производить по логарифмической шкале, а не по линейной. То есть в качестве точек, в которых считается относительная разница времени работ алгоритмов, лучше брать не $\frac{2l+r}{2}$ и $\frac{l+2r}{2}$, где l и r – текущие границы интервала поиска, а $e^{\frac{2\ln l+\ln r}{2}}$ и $e^{\ln\frac{\ln l+2\ln r}{2}}$.

Для вычисления функции относительной разницы времени работы алгоритмов троичный поиск считает ее сезнее значение по 15 запускам алгоритмов на разных случайно сгенерированных данных.

Это порождает некоторые шумы значений функции, поэтому троичный поиск не всегда может быть уверенным, что если значения функции в двух средних точках равны, то ему надо идти в средний интервал. Для того, чтобы определить, в какую сторону ему следует идти, он сравнивает разницу значений в средних точках и на границах, и идет туда, где эта разница меньше.

После нахождения размера входных данных X, на котором алгоритм BOS имеет максимальную относительную эффективность, процедура поиска границ приступает к поиску границ иинтервала эффективности BOS с помощью двоичного поиска на интервалах $[1; X]$ и $[X; 2 \cdot 10^4]$, для левой и правой границ соответственно.

Функция относительной разницы времени работы алгоритмов считается так же, как и в троичном поиске. Из-за этого появляются шумы, поэтому мы удовлетворяемся значением границы, на котором относительная разница работы абсолютно не превосходит $= 0.01$.

Описанная процедура способна достаточно быстро находить границы интервала размеров входных данных, на котором стоит запускать BOS, что необходимо для работы гибридного алгоритма.

2.2.2.3. Зависимость от оборудования

Также было замечено, что значения границ могут варьироваться от одной вычислительной машины к другой. Даже на одной машине в зависимости от ее загруженности в текущий момент эти границы могут немного сдвигаться. Это вызывает некоторые сложности из-за неактуальности уже подсчитанных границ.

2.2.2.4. Построение стратегии переключения на ходу

Предлагается в начале каждого запуска программы, которая использует алгоритм недоминирующей сортировки, производить подгонку границ. Это займет некоторое время, однако если программа будет использовать донастроенный алгоритм достаточно много раз, то итоговый выигрыш по времени может быть значительным. Если вспомнить, что недоминирующая сортировка используется в основном в алгоритмах многокритериальной оптимизации, то можно сказать, что эта ситуация возникает достаточно часто.

Также для каждого значения размерности существуют заранее посчитанные интервалы возможных значений границ, что ускоряет поиск их точных значений.

2.2.3. Модификация алгоримтма Роя

Глава 3. Практические исследования

В данной главе рассмотрены подробности реализации гибридного алгоритма, а также результаты экспериментов по сравнению реализованного алгоритма с уже существующими.

Также в данной главе приведен пример задачи, использующей недоминирующую сортировку, которая использует гибридный алгоритм, и проведено сравнение ее производительности с аналогичной реализацией задачи, использующей другой алгоритм недоминирующей сортировки.

3.1. Реализация гибридного алгоритма

В данном разделе будут описаны подробности реализации гибридного алгоритма. Будут рассмотренны основные классы, а также оптимизации, импользующиеся в данной реализации.

3.1.1. Архитектура

Тут будут описаны основные классы в реализации

3.1.2. Оптимизации в алгоритме Роя

ту будут описаны детали реализации алгоритма Роя, которые ускоряют работу этого алгоритма, а следовательно и гибридного алгоритма.

3.1.2.1. Бинарный поиск

В оригинальном алгоритме Роя используется линейный проход по массивам. Его можно заменить на бинарный поиск по элементам массива, немного ускорив его, хоть асимптотики алгоритма это и не изменит

3.1.3. Структуры данных

Еще одна оптимизация, которая не меняет асимптотики алгоритма Роя, но добивается его ускорения – использование специальных структур данных, которые заменяют структуры, описанные в оригинальной статье.

3.2. Сравнение с существующими алгоритмами на искусственно сгенерированных тестовых данных

В данном разделе описывается относительная эффективность работы алгоритма. Пока что графиков нет, но скоро будут.

3.3. Сравнение с существующими алгоритмами на практической задаче

В данном разделе будет выбрана задача оптимизации, использующая наш гибридный алгоритм, и будет проведено сравнение ее производительноси с такой же реализацией, но другим алгоритмом недоминирующей сортировки.

Заключение

Результатом данной работы будет гибридный алгоритм.

Список литературы

1. *Deb K.* A Fast Elitist Multi-Objective Genetic Algorithm: NSGA-II // Transactions on Evolutionary Computation. 2000. С. 182—197.
2. *Corne D.* The Pareto Envelope-based Selection Algorithm for Multiobjective Optimization // Parallel Problem Solving from Nature Parallel Problem Solving from Nature VI. 2000. С. 839—848.
3. *Corne D.* PESA-II: Region-based Selection in Evolutionary Multiobjective Optimization // Proceedings of Genetic and Evolutionary Com- putation Conference. 2001. С. 283—290.
4. *Zitzler E.* SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization // Proceedings of the EUROGEN'2001 Conference. 2001. С. 91—100.
5. *Knowles J.* Approximating the Nondominated Front Using the Pareto Archived Evolution Strategy // Evolutionary Computation. 2000. С. 149—172.
6. *Abbass H.* PDE: A Pareto Frontier Differential Evolution Approach for Multiobjective Optimization Problems // Proceed- ings of the Congress on Evolutionary Computation. 2001. С. 971—978.
7. *Kung H.* On Finding the Maxima of a Set of Vectors // Journal of ACM. 1975. С. 469—476.
8. *Jensen M.* Reducing the Run-time Complexity of Multiobjective EAs: The NSGA-II and Other Algorithms // Transactions on Evolutionary Computation. 2003. С. 503—515.
9. *Fortin F.* Generalizing the Improved Run-time Complexity Algorithm for Non-dominated Sorting // Proceeding of Genetic and Evolutionary Computation Conference. 2013. С. 615—622.
10. *Buzdalov M.* A Provably Asymptotically Fast Version of the Gen- eralized Jensen Algorithm for Non-Dominated Sorting // International Con- ference on Parallel Problem Solving from Nature. 2014. С. 528—537.