федеральное государственное автономное образовательное учреждение
высшего образования «Санкт-Петербургский национальный
исследовательский университет информационных технологий, механики и
оптики»

Факультет информационных технологий и программирования
Направление (специальность) Прикладная математика и информатика
Квалификация (степень) Магистр прикладной математики и информатики
Кафедра компьютерных технологий Группа 6539

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

Поддержка вычисления вспомогательных оценочных величин в структурах данных для инкрементальной недоминирующей сортировки

Автор магистерской диссертации Нигматуллин Н.Г. _____
Научный руководитель Корнеев Г.А. _____
Руководитель магистерской программы Васильев В.Н. _____

К защите допустить
Заведующий кафедрой Васильев В.Н. _____
«____» _____ 2015 г.

Санкт-Петербург, 2015 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В данной работе предлагается новый способ для улучшения удаления худшей особи в алгоритме INDS для осуществления инкрементальной недоминирующей сортировки множества двумерных точек. Структура данных, хранящая слои, теперь дополнительно разделена на блоки, в каждом из которых построена выпуклая оболочка точек, для более быстрого поиска минимума линейной функции. Оценка на время вставки одного элемента с помощью предлагаемого алгоритма равна $O(min\{N, LM\}\log L)$, где N — число точек, хранящихся в структуре данных, M — общее число недоминирующих слоев, L — выбранное значения для размера блока для выпуклой оболочки. В худшем случае это время можно оценить как $O(N\log L)$. А оценка на время удаления: $O(L + \frac{N}{L}\log L)$.

Предлагаемый алгоритм может служить одним из базовых «кирпичиков» для построения эффективных реализаций инкрементальных (англ. steady-state) многокритериальных эволюционных алгоритмов, такие как NSGA-II.

ГЛАВА 1. ОСНОВНЫЕ ПОНЯТИЯ. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

В K-мерном пространстве, точка $A = (a_1, \ldots, a_K)$ *доминирует (в смысле Парето)* точку $B = (b_1, \ldots, b_K)$, когда для всех $1 \leqslant i \leqslant K$ выполняется неравенство $a_i \leqslant b_i$, и, кроме того, существует такое j, что $a_j < b_j$. *Недоминирующая сортировка* множества точек S в K-мерном пространстве — это процедура, которая назначает всем точкам из S, которые не доминируются ни одной точкой из множества S, *ранг*, равный нулю (обозначим множество точек с нулевым рангом как S_0), всем точкам из множества $S'_0 = S \setminus S_0$, которые не доминируются ни одной точкой из множества S'_0, ранг, равный единице (множество таких точек обозначим как S_1), аналогично, всем точкам из множества $S'_i = S \setminus \bigcup_{j=0}^{i-1} S_i$, которые не доминируются ни одной точкой из множества S'_i, ранг, равный i.

Множество известных и широко применяемых 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.

В работе Кунга и др. [7] предлагается алгоритм определения множества недоминирующих точек, при этом его вычислительная сложность составляет $O(N \log^{K-1} N)$, где N — это число точек, а K — размерность пространства. Этот алгоритм возможно использовать для выполнения недоминирующей сортировки. Сначала в множестве S находятся множество всех недоминирующих точек S_0, и всем этим точкам присваивается ранг 0. Затем алгоритм Кунга запускается на оставшемся множестве точек $S \setminus S_0$, 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 $O(N^2 \log^{K-1} N)$, если максимальный ранг точки равен $\Theta(N)$.

Йенсен [8] впервые предложил алгоритм недоминирующей сортировки с вычислительной сложностью $O(N \log^{K-1} N)$. Однако, как корректность, так и

оценка сложности алгоритма доказывалась в предположении, что никакие две точки не имеют совпадающие значения ни в какой размерности. 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 [9], «обобщенный») алгоритм корректно работает во всех случаях, и во многих случаях (например, в том случае, для которого доказывалась корректность и оценка времени работы алгоритма из [8]) его время работы составляет $O(N \log^{K-1} N)$, но единственная оценка времени работы для худшего случая, доказанная в работе [8], равна $O(N^2 K)$. Наконец, в работе Буздалова и др. [10] предложены модификации алгоритма из работы [9], которые позволили доказать в худшем случае также и оценку $O(N \log^{K-1} N)$, не нарушая корректности работы алгоритма.

Одним из преимуществ эволюционных алгоритмов является возможность использования многопроцессорных и/или распределенных систем за счет параллельного вычисления функции приспособленности особей из одного поколения. Однако, синхронные варианты эволюционных алгоритмов, которые дожидаются вычисления всех особей и только после этого выполняют обновление своего вычислительного состояния, имеют ограниченное применение в распределенных системах. Даже на многопроцессорных компьютерах такие алгоритмы могут иметь сравнительно низкую производительность, если они проводят значительное время в промежутках между вычислениями функций приспособленности особей соседних поколений, так как в это время используется лишь небольшая часть ресурсов многопроцессорного компьютера. Для преодоления указанных недостатков разрабатываются инкрементальные (англ. steady-state) эволюционные алгоритмы, которые могут быть преобразованы в асинхронные алгоритмы. В частности, была разработана инкрементальная версия многокритериального алгоритма NSGA-II [11], показывающая лучшую скорость сходимости и более высокое качество аппроксимации Парето-фронта. Однако общее время работы этой версии оставляет желать лучшего.

Инкрементальная недоминирующая сортировка может быть реализована путем применения обычного алгоритма недоминирующей сортировки при каждом добавлении точки. Однако, при этом время работы становится весьма большим: $O(KN^3)$ при использовании алгоритма «быстрой недоминирующей

сортировки» из алгоритма NSGA-II [1], или $O(N^2 \log^{K-1} N)$ при использовании алгоритма из работы [10]. Следовательно, для эффективного выполнения инкрементальной недоминирующей сортировки необходимо разработать новый алгоритм.

Насколько известно исполнителю настоящей НИР, единственной работой, в которой рассматривается реализация инкрементальной недоминирующей сортировки, является научно-технический отчет авторства Ли и др. [12]. В указанном отчете предлагается процедура «Эффективное обновление недоминирующих слоев» (англ. Efficient Non-domination Level Update). Данная процедура имеет вычислительную сложность $O(NK\sqrt{N})$ (на одну операцию вставки), когда точки распределены по слоям равномерно. Экспериментальные исследования показали, что данная процедура является весьма эффективной, однако худшая оценка на время одной вставки все еще составляет $O(N^2K)$.

Во всех этих работах поиск худшей точки производился за линейное время.

В данной НИР разработан алгоритм инкрементальной недоминирующей сортировки для двумерного случая ($K = 2$), имеющий вычислительную сложность удаления $o(N)$.

1.1. Описание алгоритма для двумерного случая инкрементальной недоминирующей сортировки

В данной главе описывается алгоритм и поддерживающая его структура данных. Вид структуры данных описан в разделе 1.1.1. Процедура поиска точки (нахождения недоминирующего слоя, которому принадлежала бы данная точка после ее вставки в структуру) описана в разделе 1.1.2. Процедура вставки точки описана в разделе 1.1.3. Процедура удаления точки из последнего недоминирующего слоя описана в разделе 1.1.5.

При определении времени работы того или иного этапа алгоритма, под N понимается общее число точек, хранящихся в структуре данных, под M понимается число недоминирующих слоев. Для краткости, недоминирующие слои далее в тексте будут называться просто «слоями».

1.1.1. Структура данных

Идея, лежащая в основе структуры данных, состоит в том, чтобы хранить слои в двоичном дереве поиска в порядке возрастания соответствующего

ранга точек, так чтобы каждому узлу дерева соответствовал слой. Каждый слой, в свою очередь, также представляется в виде двоичного дерева поиска, в котором хранятся точки, отсортированные в порядке возрастания первой координаты X. Так как для двух несовпадающих точек из одного и того же слоя выполняется либо $a_X > b_X$ и $a_Y < b_Y$, либо $a_X < b_X$ и $a_Y > b_Y$, точки также получаются отсортированными в порядке убывания второй координаты Y. Псевдокод получившейся структуры данных — «дерево деревьев» — представлен на рисунке 1, а графическое представление структуры данных представлено на рисунке 2.

Отметим, что дерево, содержащее слои, может быть обычным сбалансированным деревом поиска, в то время как деревья, хранящие точки, должны быть сбалансированными деревьями поиска, разрезания и слияния. Однако, чтобы определить номер того или иного слоя за время in $O(\log M)$, а также чтобы найти за то же время, в каком слое находится t-ая по порядку точка, в вершинах дерева слоев должно храниться число слоев в поддереве, а также суммарное число точек в этом же поддереве. Также, чтобы перемещаться между соседними слоями за константное время, вершины дерева слоев должны быть дополнены указателями на следующую вершину в порядке следования, что может быть сделано без увеличения логарифмической оценки на время выполнения основных операций.

Для краткости и единообразия обозначений, дерево, содержащее в качестве элементов слои обозначим как «дерево верхнего уровня», а каждое дерево, содержащее точки, принадлежащие одному слою, как «дерево нижнего уровня».

1.1.2. Поиск слоя для вставки

Пусть дано дерево нижнего уровня T и точка s. За время $O(\log |T|)$ можно определить, доминирует ли хотя бы одна точка из T точку s. Чтобы это определить, достаточно найти такую точку $u \in T$, чтобы $u_X \leqslant s_X$ и u_X было максимально возможным. Указанная процедура может быть выполнена спуском по дереву T из его корня. Если точка u найдена и доминирует s, в этом случае доминирующая точка из T найдена, в противном случае, в T не существует точки, доминирующей s.

С использованием указанного алгоритма, можно спуститься из корня дерева верхнего уровня и определить слой с минимальным номером, который не

```
1:  structure SOLUTION
2:      — точка (решение задачи оптимизации)
3:      X — первая координата (абсцисса)
4:      Y — вторая координата (ордината)
5:  end structure
6:  structure LLTNODE
7:      — вершина дерева нижнего уровня
8:      L : LLTNODE — левый ребенок
9:      R : LLTNODE — правый ребенок
10:     V : SOLUTION — ключ упорядочения
11:     S : INTEGER — число точек в поддереве
12: end structure
13: structure HLTNODE
14:     — вершина дерева верхнего уровня
15:     L : HLTNODE — левый ребенок
16:     R : HLTNODE — правый ребенок
17:     N : HLTNODE — указатель на следующий слой
18:     V : LLTNODE — ключ упорядочения
19:     S : INTEGER — число слоев в поддереве
20:     W : INTEGER — число точек в поддереве
21: end structure
```

Рисунок 1 – Псевдокод предлагаемой структуры данных

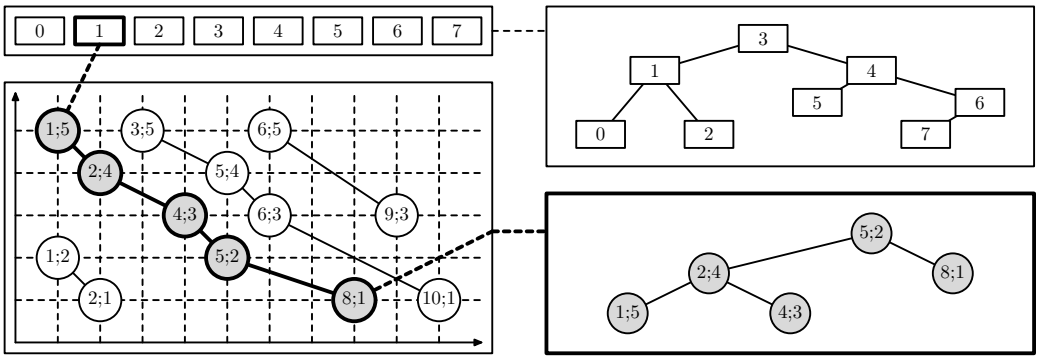

Рисунок 2 – Структура данных для алгоритма — «дерево деревьев». Вершины дерева верхнего уровня соответствуют слоям. Каждый слой, в свою очередь, представлен деревом нижнего уровня, в котором вершины соответствуют точкам и упорядочены по первой координате. Номера слоев нигде явно не хранятся, на рисунке они показаны для удобства

доминирует заданную точку s. Полученный алгоритм представлен на рисунке 3.

В работе [13] была получена оценка времени работы:

```
1: function LowLevelDominates(T, s)
2:     — определяет, существует ли точка из T, доминирующая s
3:     T : LLTNode — корень дерева нижнего уровня
4:     s : Solution — точка, проверяемая на доминирование
5:     B ← NULL — лучшая найденная вершина
6:     while T ≠ NULL do
7:         if T.V.X ⩽ s.X then
8:             B ← T
9:             T ← T.R
10:        else
11:            T ← T.L
12:        end if
13:    end while
14:    if B = NULL then
15:        return FALSE
16:    end if
17:    return B.Y < s.Y or B.Y = s.Y and B.X < s.X
18: end function
19: function SmallestNonDominatingLayer(H, s)
20:    — возвращает слой с минимальным индексом из H
21:    — такой что ни одна его вершина не доминирует s
22:    H : HLTNode — корень дерева верхнего уровня
23:    s : Solution — точка, для которой ищется слой
24:    I ← 0 — число слоев, которые доминируют точку
25:    B ← NULL — лучший из найденных слоев
26:    while H ≠ NULL do
27:        if LowLevelDominates(H.V, s) then
28:            I ← I + H.S
29:            H ← H.R
30:            if H ≠ NULL then
31:                I ← I − H.S
32:            end if
33:        else
34:            B ← H
35:            H ← H.L
36:        end if
37:    end while
38:    return (B, I)
39: end function
```

Рисунок 3 – Псевдокод для определения слоя с наименьшим индексом, в котором нет точек, доминирующих данную

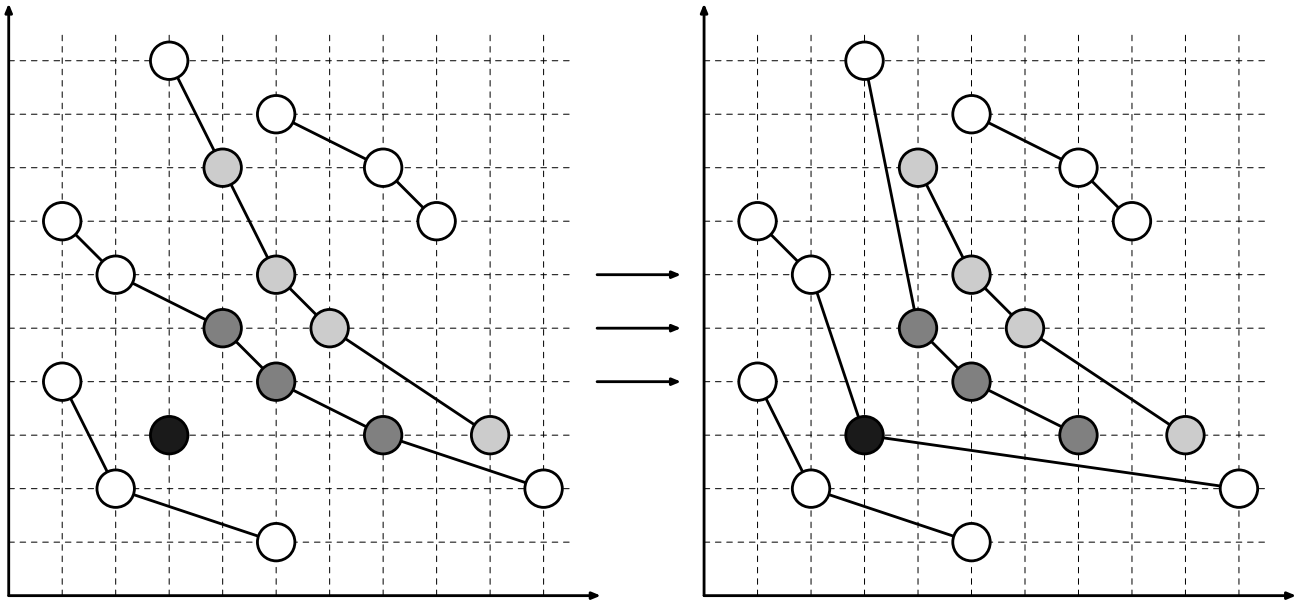

Рисунок 4 – Иллюстрация процесса вставки. Точки, для которых слой (но не его номер) после вставки не изменился, обозначены белым. Точка, которая вставляется в структуру, обозначена черным. Две группы решений, которые совместно перемещаются из слоя в слой, обозначены соответственно темно-серым и светло-серым

$$O\left(\log M \log \frac{N}{\log M}\right).$$

1.1.3. Вставка

Пусть дано дерево верхнего уровня H и точка s, процедура вставки обновляет дерево H таким образом, что точка s оказывается в одном из его деревьев нижнего уровня.

Ключевая идея быстрой реализации процедуры вставки состоит в том, что точки, меняющие номер слоя, в котором они находятся, образуют непрерывные подпоследовательности в своих исходных слоях и остаются непрерывными также и в своих новых слоях. На рисунке 4 дан пример процесса вставки.

Псевдокод алгоритма для вставки точки представлен на рисунке 5. Алгоритм поддерживает дерево нижнего уровня, которое на каждой итерации алгоритма содержит точки, которые требуется вставить в следующий слой. В начале работы это дерево содержит единственную точку — ту точку, которую требуется вставить в структуру. Слой, в который нужно вставить эту точку, определяется с помощью процедуры SMALLESTNONDOMINATINGLAYER (рисунок 3).

```
1: function SplitX(T, s)
2:     — разрезает дерево T на две части L, R, так что для всех l ∈ L
3:     — выполняется l.X < s.X и для всех r ∈ R выполняется r.X ⩾ s.X
4:     T : LLTNode
5:     s : Solution
6: end function
7: function SplitY(T, s)
8:     — разрезает дерево T на две части L, R, так что для всех l ∈ L
9:     — выполняется l.Y ⩾ s.Y и для всех r ∈ R выполняется r.Y < s.Y
10:    T : LLTNode
11:    s : Solution
12: end function
13: function Merge(L, R)
14:    — сливает L и R в одно дерево в предположении, что
15:    — для всех l ∈ L и r ∈ R выполняется l.X < r.X
16:    L, R : LLTNode
17: end function
18: function Insert(H, s)
19:    – inserts a solution s into a high-level tree H
20:    H : HLTNode
21:    s : Solution
22:    C ← new LLTNode(s)
23:    (G, i) ← SmallestNonDominatingLayer(H, s)
24:    while G ≠ null do
25:        C_min ← точка с минимальным x из C
26:        C_max ← точка с минимальным y из C
27:        (T_L, T_i) ← SplitX(G.V, C_min)
28:        (T_M, T_R) ← SplitY(T_i, C_max)
29:        G.V ← Merge(T_L, Merge(C, T_R))
30:        if T_M = null then
31:            return — больше нет решений для проталкивания
32:        end if
33:        if T_L = null and T_R = null then
34:            Вставить new HLTNode(T_M) после G
35:            return
36:        end if
37:        C ← T_M, G ← G.N
38:    end while
39:    Вставить new HLTNode(C) после последней вершины H
40: end function
```

Рисунок 5 – Псевдокод процедуры вставки точки в дерево высокого уровня

Алгоритм совершает итерации, на каждой итерации он проталкивает текущий набор точек в слой, который непосредственно доминируется слоем, над которым работала предыдущая итерация. На каждой итерации выполняются следующие операции:

— Дерево нижнего уровня T, хранящее текущий слой, разбивается на три части с использованием текущего проталкиваемого множества точек C следующим образом:
  – «левая» часть T_L состоит из тех точек текущего слоя, чьи абсциссы меньше, чем наименьшая абсцисса какой-либо точки из множества C;
  – «средняя» часть T_M состоит их тех точек текущего слоя, которые доминируются хотя бы одной точкой из множества C;
  – «правая» часть T_R состоит из тех точек текущего слоя, чьи ординаты меньше, чем наименьшая ордината какой-либо точки из множества C.

  Корректность указанного разбиения будет доказана ниже в Лемме **??**.
— Текущий уровень собирается путем слияния деревьев T_L, C и T_R.
— Если оба множества T_L и T_R пусты, это означает, что весь текущий уровень доминируется решениями из множества C. В свою очередь, это означает, что новый слой, состоящий из точек множества T_M, должен быть вставлен сразу после текущего уровня. Все остальные уровни просто увеличивают свой номер на единицу, поэтому процедура вставки окончена.
— Если T_M пусто, остальные слои не будут изменены. Процедура вставки окончена.
— В остальных случаях, $C \leftarrow T_M$, и процедура вставки продолжается на следующей итерации.

Если после выполнения последней итерации остаются точки, которые никуда не вставлены, из них формируется новый уровень, который добавляется в дерево верхнего уровня в качестве самого последнего уровня.

Общая сложность алгоритма вставки составляет:

$$O\left(M\left(1+\log\frac{N}{M}\right)+\log M\log\frac{N}{\log M}\right),$$

что в худшем случае $O(N)$.

1.1.4. Поиск k-той точки в порядке следования

В алгоритме NSGA-II используется оператор выбора, основанный на состязании двух особей, который рассматривает каждую особь ровно два раза. Данный оператор поддерживает перестановку индексов $1 \dots N$, где N — размер поколения. Когда требуется выбрать особь из поколения, оператор берет элемент E указанной перестановки, который еще не был выбран, и возвращает особь, находящуюся в поколении на позиции E. Когда вся перестановка исчерпана, генерируется новая случайная перестановка и процесс повторяется.

Для поддержки указанного оператора, в разрабатываемом алгоритме требуется возвращать k-ую точку, вместе с ее рангом, в некотором предопределенном порядке. Обратим внимание, что взятие случайной точки может быть реализовано с помощью операции взятия k-ой точки, если k выбирать случайно из отрезка $[1; N]$. Каждый запрос возвращает как точку, так и ее ранг.

Для реализации описанной операции, будем использовать хранимые в вершинах дерева верхнего уровня размер поддерева и общее число точек во всех слоях из поддерева, а также хранимые в вершинах деревьев нижнего уровня размеры поддерева. Будем использовать в некотором смысле лексикографический порядок точек — сначала для сравнения точек используются их ранги (номер слоя), а при равенстве рангов — их абсциссы.

Пусть дано число k — порядковый номер точки в указанном выше порядке. Первая часть алгоритма поиска k-ой точки находит номер слоя, в котором находится данная точка, что осуществляется путем спуска от корня в дереве высокого уровня. Вторая часть алгоритма осуществляет ту же процедуру, но в выбранном дереве низкого уровня. Время работы указанного алгоритма составляет $O(\log N)$.

1.1.5. Удаление худшей точки

В большинстве многокритериальных алгоритмов удаление произвольной точки не требуется, и, следовательно, его не обязательно поддерживать. Единственные точки, которые могут удаляться, это «худшие» точки, принадлежащие последнему слою, которые могут быть удалены без перестройки всей структуры данных. Время выполнения произвольной точки с последнего слоя составляет $O(\log N + \log M)$, при этом слагаемое $O(\log M)$ соответствует случаю, когда требуется удалить последний слой целиком.

1.1.6. Поиск худшей точки

В алгоритме NSGA-II используется вспомогательная оценочная величина для определения худшей точки, она называется *плотность заселения* (в английской литературе используется термин *crowding distance*)[1]. Это способ измерить, насколько особь близка к своим соседям. Чем больше среднее значение плотности заселения, тем более разнообразна популяция. Один из способов выбирать «худшую» точку — находить точку с минимальной плостностью заселения в последнем слое.

Плотность заселения вычисляется по каждому критерию независимо: для каждой особи выбираются два соседних по этому критерию, и вычисляется сумма нормированных разностей по всем критериям.

Определим более формально:

- Пусть в слое F_i есть n особей и $d(F_i(j))$ — плотность заселения для особи $F_i(j)$
  - Для каждого критерия k: R — отсортированный список особей по k-му критерию $f(R_i, k)$
  - $d_k(R_1)$ и $d_k(R_n)$ положим равными бесконечности
  - $d_k(R_j) = \frac{f(R_{j+1},k)-f(R_{j-1},k)}{f(R_n,k)-f(R_1,k)}$, для $1 < j < n$
- $d(F_i(j)) = \sum\limits_k d_k(F_i(j))$

Для двумерного случая соседние особи по обоим критериям одинаковые. Плотность заселения запишем как:

$$d(R_j) = \frac{x_j}{X} + \frac{y_j}{Y}$$

где x_j и y_j — это значение $(f(R_{j+1}, k) - f(R_{j-1}, k))$, а $\frac{1}{X}$ и $\frac{1}{Y}$ — значение $(f(R_n, k) - f(R_1, k))$ для первого и второго критериев соответственно.

В работе [13] поиск особи с минимальной плотностью заселения производится за линейное время.

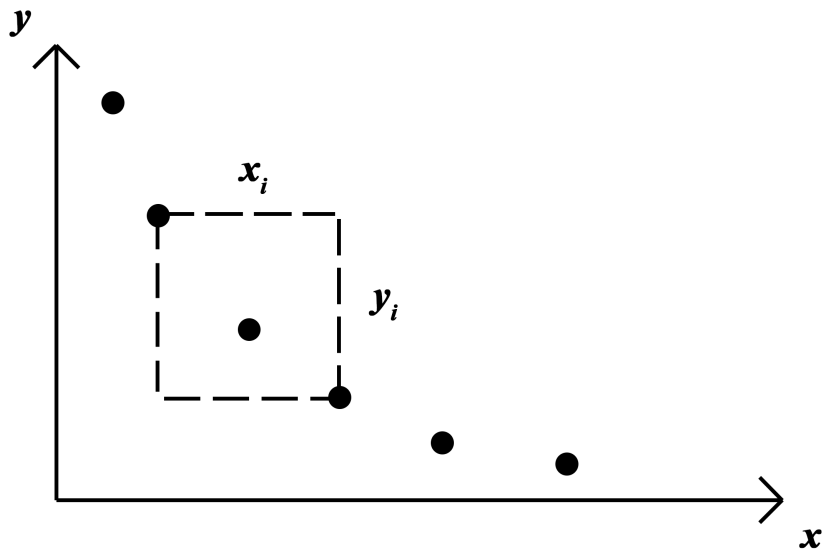

Рисунок 6

ГЛАВА 2. СВЕДЕНИЕ ЗАДАЧИ О ПОИСКЕ ХУДШЕЙ ТОЧКИ К ПОСТРОЕНИЮ ВЫПУКЛОЙ ОБОЛОЧКИ И МИНИМИЗАЦИИ ЛИНЕЙНОЙ ФУНКЦИИ

В попытке улучшить алгоритм поиска худшей точки было бы очень полезно поддерживать плотность заселения каждой особи и выбирать минимум, используя известные структуры данных. Однако, явно хранить эту величину не представляется возможным, потому что при добавлении новых точек с максимальными или минимальными значениями функции приспособленности в слой, нормировочный коэффициент

$$\frac{1}{f(R_n, k) - f(R_1, k)}$$

изменяется, вместе с этим меняется значение плотность заселения для всех точек в этом слое. Однако, число точек, у которых поменяются соседи, а вместе с ними и разность функций приспособленности соседей

$$f(R_{j+1}, k) - f(R_{j-1}, k),$$

небольшое: при добавление непрерывной подпоследовательности в новый слой, соседи поменяются у $O(1)$ точек. Из этого был сделан вывод, что явно можно хранить только разности соседей: в двумерном случае x_j и y_j. А нормировочные коэффициенты: $\frac{1}{X}$ и $\frac{1}{Y}$ могут изменяться часто, это надо учесть.

Поиск такой R_j, что $d(R_j)$ минимальна, можно свести к задаче:

— Задано множество точек (x_j, y_j), в нашей задаче x_j и y_j — разности соседей особи R_j;
— Нужно отвечать на запросы:
  – Параметры запроса — два числа: a и b, в нашей задаче $a = \frac{1}{X}$ и $b = \frac{1}{Y}$;
  – Найти такую точку i в заданном множестве, что $ax_i + by_i$ минимально.

Точка, в которой $ax + by$ минимально, будет лежать на границе выпуклой оболочки точек множества, более того, одна из вершин оболочки будет оптимальной. Если из точки (x, y) сдвинуться по вектору, скалярное произведение которого с вектором (a, b) отрицательно, то значение функции уменьшится. Поэтому оптимальной будет точка, что по одну сторону от прямой, перпен-

дикулярной (a, b) точек из множества нет. А это всегда граница выпуклого многоугольника.

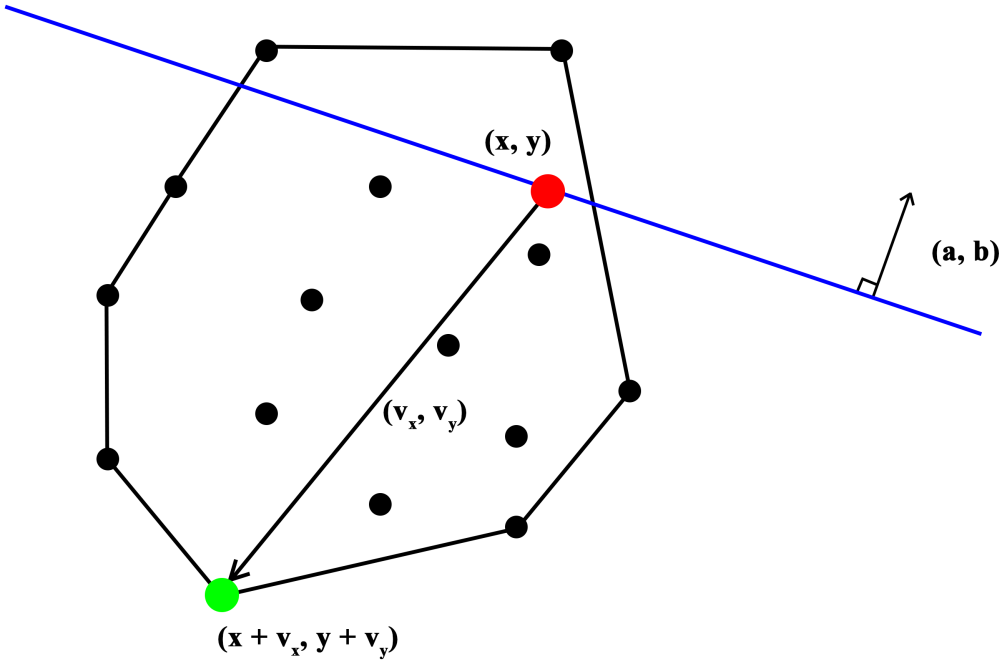

Рисунок 7

Существует алгоритм, основанный на двоичном поиске по выпуклой оболочке, который за время $O(\log N)$ (где, N — число вершин выпуклой оболочки) находит точку, где достигается минимум линейной функции. Алгоритм построения выпуклой оболочки и алгоритм поиска точки с минимальным значением линейной функции в ней описан в книге [14].

ГЛАВА 3. АЛГОРИТМ, ИСПОЛЬЗУЮЩИЙ ВЫПУКЛУЮ ОБОЛОЧКУ ДЛЯ ИНКРЕМЕНТАЛЬНОЙ НЕДОМИНИРУЮЩЕЙ СОРТИРОВКИ

3.1. Общая идея применения выпуклой оболочки в алгоритме INDS

Главной идеей предлагаемых алгоритмов будет объединение точек в группы и построения для этих точек выпуклой оболочки, чтобы в дальнейшем находить точку с минимальным значением плотности заселения, за время $O(\log L)$ для этой группы из L точек, что потенциально даст улучшение во времени поиска в $\frac{L}{\log L}$ раз.

Так как поиск худшей точки всегда будет производиться в самом последнем слое, точки в группы будут объединятся только из одного слоя. Сложность задачи в том, что слои меняются, и нужно группы и их выпуклые оболочки поддерживать в актуальном состоянии после каждого добавления и удаления. Так как в алгоритме INDS точки переходят из слоя в слой непрерывными подпоследовательностями, то выпуклые оболочки тоже разумно строить для непрерывных подпоследовательностей в одном слое.

Структура `ConvexHull` будет поддерживать следующие операции:

— Построение структуры по первой и последней точке в подпоследовательности

— Запрос на худшую точку с параметрами a и b

— При изменении подпоследовательности, для которой построена выпуклая оболочка, выставлять флаг, который обозначает, что структура более не актуальна

Также структура будет строить выпуклую оболочку только при первом запросе на худшую точку, потому что, запросы будут приходить только для последнего слоя, и при некоторых реализациях алгоритмов к некоторым экземлярам структуры не придет ни единого запроса.

Каждая точка, которой соответствует экземпляр `LLTNode`, будет либо находится в какой-то группе, для которой построена выпуклая оболочка, либо нет. При первом варианте, в ней будет храниться ссылка на соответствующую структуру `ConvexHull`.

Заметим, что точка (x_j, y_j) меняется для особи, когда у нее меняются соседи.

```
1: structure ConvexHull
2:     — выпуклая оболочка для поиска оптимальной точки
3:     L — первая точка группы, для которой строится выпуклая оболочка
4:     R — последняя точка группы, для которой строится выпуклая оболочка
5:     H — упорядоченный для поиска массив точек на выпуклой оболочке
6: end structure
```

Рисунок 8 – Псевдокод предлагаемой структуры данных для хранения выпуклой оболочки

3.2. Построение выпуклых оболочек на каждом слое

Рассмотрим первый способ улучшить удаление худшей точки в алгоритме INDS с помощью идеи выпуклой оболочки и структуры `ConvexHull`. В алгоритме выбирается число L и поддерживается следующий инвариант:

- Если точка находится в слое, размер которого меньше $\frac{L}{2}$, то точка не входит ни в какую группу, и для нее выпуклая оболочка не построена
- Если точка находится в слое, размер которого не меньше $\frac{L}{2}$, то она входит в группу, размер которой от $\frac{L}{2}$ до $2L$

Заметим, что любой такой слой всегда можно разбить на группы размером от $\frac{L}{2}$ до $2L$.

3.2.1. Поддержка инварианта при операциях изменения слоев

При изменении слоя в алгоритме INDS множество точек на этом слое, для которых выпуклая оболочка становится неактуальной, всегда образует одну или две непрерывные подпоследовательности. Нужно научиться восстанавливать инвариант при перестроении. При вырезании подпоследовательности из слоя, актуальными перестают быть две группы, соседствующие с границами вырезания. При вставке туда же подпоследовательности из предыдущего слоя, две группы по краям также имеют неактуальные выпуклые оболочки. На каждой из этих подпоследовательностей, на которых выпуклая оболочка не актуальна, не больше $4L$ точек (или не больше $8L$ точек, если подпоследовательность одна). Все эти точки, мы можем разбить на $O(1)$ групп так, что их размеры будут не больше $2L$ и размеры всех, кроме может быть одной группы, будут не меньше $\frac{L}{2}$. Если в слое группа с размером $s_0 < \frac{L}{2}$, то возникает два случая:

- Если в слое больше нет групп, то выпуклую оболочку строить не надо, первое условие инварианта выполняется;

— Если в слое есть еще группы, то выберем какую-нибудь соседнюю для группы с размером s_0, пусть ее размер s_1. Известно, что $\frac{L}{2} \leqslant s_1 \leqslant 2L$. Рассмотрим два случая:

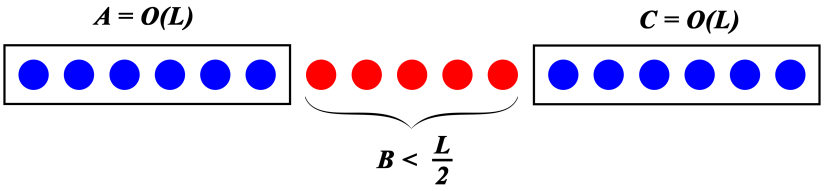

Рисунок 9

– Если $s_0 + s_1 \leqslant 2L$, тогда мы знаем, что $s_0 + s_1 \geqslant \frac{L}{2}$, потому что $s_1 \geqslant \frac{L}{2}$. В этом случае объединим эти две группы, инвариант сохранился;

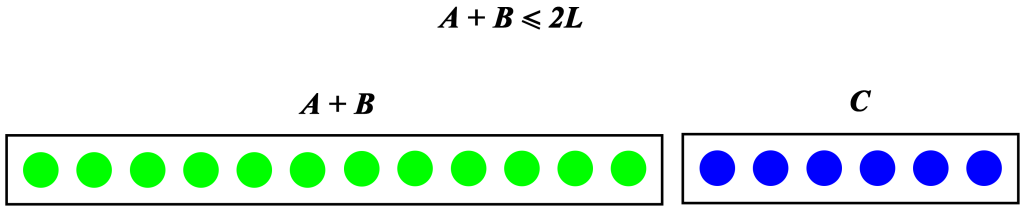

Рисунок 10

– Если $s_0 + s_1 > 2L$, то можно все эти точки разбить на две группы размером не меньше L и не больше $2L$, так как $s_0 + s_1 \leqslant 4L$. Инвариант так же сохранился.

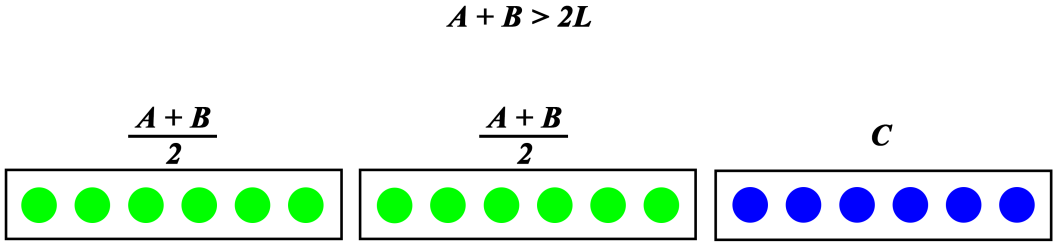

Рисунок 11

3.2.2. Оценка времени работы добавления

Оценим дополнительное время работы для поддержки выпуклых оболочек с заданным инвариантом для операции вставки в алгоритме INDS. Для слоев, в которых меньше $\frac{L}{2}$ точек, никаких перестроений групп и выпуклых оболочек не происходит, а для остальных слоев, перестраивается $O(1)$ выпуклых оболочек. Так как все перестроения происходят лениво, то во всех слоях кроме последнего выпуклая оболочка строится за $O(L \log L)$. Суммарно имеем:

$$O\left(\log L \sum_{k=1}^{M} \min\{L, n_k\} + L \log L\right) = O(min\{LM, N\} \log L).$$

При числе слоев $M < \frac{N}{L \log L}$ добавление точки в худшем случае работает за $O(N)$.

3.2.3. Оценка времени работы поиска и удаления

Для поиска худшей точки:

— Если размер последнего слоя меньше $\frac{L}{2}$, то поиск проведем за размер слоя, то есть в худшем случае за $O(L)$;

— Иначе, процесс поиска худшей точки выполняется за время $O(\frac{S}{L} \log L)$. В каждой из $O(\frac{S}{L})$ групп за $O(\log L)$.

Удаление худшей точки производится за $O(L + \log N)$.

В итоге, при выборе $L = O(\sqrt{N})$ имеем $O(N)$ на вставку, если $M < \frac{\sqrt{N}}{\log N}$, и $O(\sqrt{N} \log N)$ на удаление. Заметим, что в эволюционных алгоритмах число слоев обычно бывает не очень большим, часто меньше $\frac{\sqrt{N}}{\log N}$, поэтому на практике добавление остается линейным.

3.3. Построение выпуклых оболочек только на последнем слое

Так как запросы поступают только к последнему слою, было бы логично не строить выпуклые оболочки и не разбивать на группы точки на других слоях.

3.3.1. Ленивый алгоритм построения выпуклых оболочек

Модифицируем алгоритм так, что точки не всегда объединены в группы, то есть некоторые точки могут быть вне групп. Все группы, которые есть, они также как и раньше должны быть размера не меньше $\frac{L}{2}$ и не больше $2L$. Каждый раз, когда у точки меняются соседи, группа, в которой эта точка была, разрушается.

Перед каждым удалением происходит пересчет выпуклых оболочек для последнего слоя. После этого пересчета, инвариант должен быть выполнен для последнего слоя. Алгоритм пересчета устроен так:

— Выделим все непрерывные подпоследовательности из точек, которые не относятся к какой-либо группе
— Каждую подпоследовательность можно разделить на группы размерами не меньше $\frac{L}{2}$ и не больше $2L$, кроме, может быть одной маленькой группы. А эту маленькую группу можно объединить с соседней, как в предыдущем алгоритме.

3.3.2. Время работы

Докажем, что дополнительно время работы на Q_i вставок и Q_r удалений худшей точки будет работать за время

$$O\left(Q_i(\min\{N, LM\}\log L) + Q_r\left(L + \frac{S}{L}\log L\right)\right)$$

Применим метод потенциалов, введем потенциальную функцию:

$$\Phi(Q_i, Q_r) = \sum_{i=1}^{N} \phi(Q_i, Q_r, v_i),$$

где $\phi(Q_i, Q_r, v)$ — потенциал вершины v, который определим, как:

— Если для вершины v никогда не строилась выпуклая оболочка, то $\phi(Q_i, Q_r, v) = \log N$
— Если для вершины v строилась выпуклая оболочка, но у вершины v поменялся сосед и после этого выпуклая оболочка более не перестраивалась, то $\phi(Q_i, Q_r, v) = L\log L$
— Если для вершины v есть выпуклая оболочка, то $\phi(Q_i, Q_r, v) = 0$
— Если для вершины v построилась выпуклая оболочка, но соседи с этого момента не поменялись, то $\phi(Q_i, Q_r, v) = 0$

При вставке на каждом слое увеличится потенциал у $O(1)$ вершин, а значит, по аналогии доказательства времени вставки в первом алгоритме:

$$\Phi(Q_i, Q_r) = \Phi(Q_i - 1, Q_r) + O(\log L \min\{LM, N\} + \log N).$$

При удалении потенциал вершин не увеличивается:

$$\Phi(Q_i, Q_r) \leqslant \Phi(Q_i, Q_r - 1).$$

Из этого делаем вывод, что

$$\Phi(Q_i, Q_r) = O(Q_i \min\{LM, N\} \log L)$$

Рассмотрим среднее время на вставку:

$$a_{ins}(i) = \Phi(Q_i, Q_r) - \Phi(Q_i - 1, Q_r) + t_{ins}(i).$$

Истинное дополнительное время работы добавления $t_{ins}(i) = 0$, а значит

$$a_{ins}(i) = O(\log L \min\{LM, N\})$$

Рассмотрим среднее время на удаление:

$$a_{del}(i) = \Phi(Q_i, Q_r) - \Phi(Q_i, Q_r - 1) + t_{del}(i).$$

Если точек в последнем слое хотя бы $\frac{L}{2}$, то истинное время удаления $t_{del}(i) = O(L + \frac{S}{L} \log L + B \log L)$, где B — число точек, для которых не была посчитана выпуклая оболочка. Посчитаем насколько уменьшится потенциал после удаления точки. Точки, для которых мы еще ни разу не строили выпуклые оболочки вносят вклад $O(\log L)$ каждая, пусть, таких точек B_0. Рассмотрим непрерывный отрезок точек, для которых мы уже строили выпуклые оболочки, но в данный момент не относятся ни к какой группе. У каждой из таких точек потенциал либо 0, либо $O(L \log L)$. Заметим, что:

1. на любом таком максимальном по включению отрезке существует точка, у которой потенциал ненулевой;
2. не существует отрезка длины больше $2L$, такого что, потенциалы точек на этом отрезке все равны 0.

Первое верно, потому что существует какая-то точка v_0, для которой оболочка бва ыла построена, а теперь ее нет. Можно рассмотреть самую ближайшую точку справа и самую ближайшую точку слева, которая не из этой же группы что и точка v_0, либо которая была в другом слое после развала груп-

пы. Либо правая, либо левая существует, потому что иначе группа бы осталась существовать, и соседняя с правой либо левой, меняла соседа, а значит потенциал у нее $O(L \log L)$. Второе верно по той же причине, так как группы не бывают размером больше $2L$, значит на расстоянии не более $2L$ всегда найдется точка с потенциалом $O(L \log L)$. А это значит, что потенциал уменьшается на $(B_0 + B_1) \log L = O(B \log L)$. Из чего делаем вывод, что

$$a_{del}(i) = O\left(L + \frac{S}{L} \log L\right).$$

Итого, мы доказали, что Q_i вставок и Q_r удалений работают за:

$$O\left(Q_i(\min\{N, LM\} \log L) + Q_r\left(L + \frac{S}{L} \log L\right)\right),$$

что не хуже, чем в предыдущем алгоритме. В этом алгоритме мы имеем теоретическое преимущество в практическом применении, так как все операции, которые мы делаем для построения выпуклой оболочки, откладываются на последний слой и на последний момент. Некоторые из тех построений групп размера порядка L и построений выпуклых оболочек могут вообще не понадобится.

ГЛАВА 4. ЭКСПЕРИМЕНТАЛЬНОЕ ТЕСТИРОВАНИЕ АЛГОРИТМА

Обе версии алгоритма были реализованы на языке программирования Java.

Листинг 1 – Реализация восстановления инварианта в построении выпуклых оболочек на всех слоях

```
@Override
protected void llNodeLinkHook(LLNode node, boolean isSetPrev) {
    int upperLimit = maxLastLayerSize = Math.max(
       maxLastLayerSize, (int) (Math.sqrt(layerRoot.rightmost()
       .key().size())));
    if (isSetPrev) {
        LLNode prev = node.prev();
        if (prev == null) {
            new ConvexHull(node, node.hull.last);
        } else if (prev.next() == node) {
            if (node.hull == null || prev.hull == null || node.
               hull != prev.hull) {
                int sumRange = (node.hull == null ? 0 : node.
                   hull.rangeSize) + (prev.hull == null ? 0 :
                   prev.hull.rangeSize);
                LLNode theFirst = prev.hull == null ? prev :
                   prev.hull.first;
                LLNode theLast = node.hull == null ? node :
                   node.hull.last;
                if (sumRange <= upperLimit) {
                    new ConvexHull(theFirst, theLast);
                } else {
                    LLNode mid = theFirst;
                    for (int i = 2; i < sumRange; i += 2) {
                        mid = mid.next();
                    }
                    new ConvexHull(theFirst, mid);
                    new ConvexHull(mid.next(), theLast);
                }
            }
        }
    } else {
        LLNode next = node.next();
        if (next == null) {
            new ConvexHull(node.hull.first, node);
```

```
            } else if (next.prev() == node) {
                if (node.hull == null || next.hull == null || node.
                    hull != next.hull) {
                    int sumRange = (node.hull == null ? 0 : node.
                        hull.rangeSize) + (next.hull == null ? 0 :
                        next.hull.rangeSize);
                    LLNode theFirst = node.hull == null ? node :
                        node.hull.first;
                    LLNode theLast = next.hull == null ? next :
                        next.hull.last;
                    if (sumRange <= upperLimit) {
                        new ConvexHull(theFirst, theLast);
                    } else {
                        LLNode mid = theFirst;
                        for (int i = 2; i < sumRange; i += 2) {
                            mid = mid.next();
                        }
                        new ConvexHull(theFirst, mid);
                        new ConvexHull(mid.next(), theLast);
                    }
                }
            }
        }
    }
```

Листинг 2 – Построение выпуклых оболочек только на последнем слое

```
    @Override
    protected LLNode removeWorstByCrowding(int count) {
        int upperLimit = maxLastLayerSize = Math.max(
            maxLastLayerSize, (int) (Math.sqrt(1. * layerRoot.
            rightmost().key().size() / layerRoot.size())));
        if (currentUpperLimit * 2 < upperLimit) {
            currentUpperLimit = upperLimit;
            rebuildAll(currentUpperLimit);
        } else {
            rebuildLastLayer(currentUpperLimit);
        }
        return super.removeWorstByCrowding(count);
    }

    private void rebuildAll(int upperLimit) {
```

```
        if (layerRoot == null) return;
        LLNode start = layerRoot.rightmost().key().leftmost();
        for (LLNode first = start; first != null; first = first.
           next()) {
            first.hull = null;
        }
        rebuildLastLayer(upperLimit);
    }

    private void rebuildLastLayer(int upperLimit) {
        if (layerRoot == null) return;
        LLNode start = layerRoot.rightmost().key().leftmost();
        for (LLNode first = start; first != null; ) {
            if (first.hull != null && first.hull.isAlive) {
                first = first.hull.last.next();
                continue;
            }
            LLNode last = first;
            int cnt = 1;
            while (last.next() != null && (last.next().hull == null
                || !last.next().hull.isAlive)) {
                last = last.next();
                ++cnt;
            }
            if (2 * cnt < upperLimit) {
                LLNode prev = first.prev();
                LLNode next = last.next();
                if (prev != null) {
                    int prevSize = prev.hull.rangeSize;
                    first = prev.hull.first;
                    cnt += prevSize;
                } else if (next != null) {
                    int nextSize = next.hull.rangeSize;
                    last = next.hull.last;
                    cnt += nextSize;
                } else return;
            }
            while (cnt > 0) {
                int get = cnt >= 2 * upperLimit ? upperLimit : cnt;
                LLNode cur = first;
                cnt -= get;
```

```
                --get;
                while (get > 0) {
                    get--;
                    cur = cur.next();
                }
                new ConvexHull(first, cur);
                first = cur.next();
            }
        }
    }
```

Тестирование проводилось на нескольких задачах для эволюционных алгоритмов. Для каждой из задач было несколько категорий тестов с различным числом особей в поколении и различным числом итераций эволюционного алгоритма. Далее представлены таблицы с результатами тестирования.

Таблица 1 – Время работы. Размер поколения 100, число итераций 25000

Problem	INDS	INDS LastHull	INDS AllHulls
DTLZ1	$4.106 \cdot 10^{-2}$	$4.986 \cdot 10^{-2}$	$5.847 \cdot 10^{-2}$
DTLZ2	$5.751 \cdot 10^{-2}$	$6.807 \cdot 10^{-2}$	$7.566 \cdot 10^{-2}$
DTLZ3	$3.895 \cdot 10^{-2}$	$4.668 \cdot 10^{-2}$	$5.990 \cdot 10^{-2}$
DTLZ4	$5.701 \cdot 10^{-2}$	$6.597 \cdot 10^{-2}$	$7.255 \cdot 10^{-2}$
DTLZ5	$5.653 \cdot 10^{-2}$	$6.794 \cdot 10^{-2}$	$7.501 \cdot 10^{-2}$
DTLZ6	$6.037 \cdot 10^{-2}$	$6.789 \cdot 10^{-2}$	$8.518 \cdot 10^{-2}$
DTLZ7	$5.721 \cdot 10^{-2}$	$6.921 \cdot 10^{-2}$	$7.986 \cdot 10^{-2}$
WFG1	$5.238 \cdot 10^{-2}$	$6.296 \cdot 10^{-2}$	$7.026 \cdot 10^{-2}$
WFG2	$8.182 \cdot 10^{-2}$	$9.939 \cdot 10^{-2}$	$1.068 \cdot 10^{-1}$
WFG3	$6.167 \cdot 10^{-2}$	$7.213 \cdot 10^{-2}$	$7.752 \cdot 10^{-2}$
WFG4	$5.948 \cdot 10^{-2}$	$7.317 \cdot 10^{-2}$	$7.849 \cdot 10^{-2}$
WFG5	$7.739 \cdot 10^{-2}$	$9.212 \cdot 10^{-2}$	$9.980 \cdot 10^{-2}$
WFG6	$5.805 \cdot 10^{-2}$	$6.920 \cdot 10^{-2}$	$7.440 \cdot 10^{-2}$
WFG7	$6.712 \cdot 10^{-2}$	$7.933 \cdot 10^{-2}$	$8.540 \cdot 10^{-2}$
WFG8	$3.425 \cdot 10^{-2}$	$3.907 \cdot 10^{-2}$	$4.282 \cdot 10^{-2}$
WFG9	$6.304 \cdot 10^{-2}$	$7.492 \cdot 10^{-2}$	$8.112 \cdot 10^{-2}$
ZDT1	$5.663 \cdot 10^{-2}$	$7.200 \cdot 10^{-2}$	$8.440 \cdot 10^{-2}$
ZDT2	$5.818 \cdot 10^{-2}$	$7.143 \cdot 10^{-2}$	$8.751 \cdot 10^{-2}$
ZDT3	$5.606 \cdot 10^{-2}$	$6.842 \cdot 10^{-2}$	$8.312 \cdot 10^{-2}$
ZDT4	$3.922 \cdot 10^{-2}$	$4.714 \cdot 10^{-2}$	$5.790 \cdot 10^{-2}$
ZDT6	$5.646 \cdot 10^{-2}$	$6.802 \cdot 10^{-2}$	$8.124 \cdot 10^{-2}$

Таблица 2 – Число сравнений. Размер поколения 100, число итераций 25000

Problem	INDS	INDS LastHull	INDS AllHulls
DTLZ1	$2.306 \cdot 10^6$	$2.146 \cdot 10^6$	$1.559 \cdot 10^6$
DTLZ2	$4.138 \cdot 10^6$	$3.718 \cdot 10^6$	$2.316 \cdot 10^6$
DTLZ3	$1.360 \cdot 10^6$	$1.357 \cdot 10^6$	$1.337 \cdot 10^6$
DTLZ4	$3.917 \cdot 10^6$	$3.508 \cdot 10^6$	$2.212 \cdot 10^6$
DTLZ5	$4.141 \cdot 10^6$	$3.719 \cdot 10^6$	$2.322 \cdot 10^6$
DTLZ6	$3.635 \cdot 10^6$	$3.307 \cdot 10^6$	$2.317 \cdot 10^6$
DTLZ7	$3.544 \cdot 10^6$	$3.270 \cdot 10^6$	$2.227 \cdot 10^6$
WFG1	$3.169 \cdot 10^6$	$2.874 \cdot 10^6$	$1.946 \cdot 10^6$
WFG2	$7.033 \cdot 10^6$	$6.152 \cdot 10^6$	$3.550 \cdot 10^6$
WFG3	$4.898 \cdot 10^6$	$4.295 \cdot 10^6$	$2.507 \cdot 10^6$
WFG4	$4.735 \cdot 10^6$	$4.163 \cdot 10^6$	$2.483 \cdot 10^6$
WFG5	$6.653 \cdot 10^6$	$5.808 \cdot 10^6$	$3.343 \cdot 10^6$
WFG6	$4.436 \cdot 10^6$	$3.898 \cdot 10^6$	$2.377 \cdot 10^6$
WFG7	$5.474 \cdot 10^6$	$4.791 \cdot 10^6$	$2.795 \cdot 10^6$
WFG8	$1.586 \cdot 10^6$	$1.498 \cdot 10^6$	$1.137 \cdot 10^6$
WFG9	$4.836 \cdot 10^6$	$4.281 \cdot 10^6$	$2.522 \cdot 10^6$
ZDT1	$3.741 \cdot 10^6$	$3.446 \cdot 10^6$	$2.324 \cdot 10^6$
ZDT2	$3.326 \cdot 10^6$	$3.085 \cdot 10^6$	$2.198 \cdot 10^6$
ZDT3	$3.395 \cdot 10^6$	$3.159 \cdot 10^6$	$2.223 \cdot 10^6$
ZDT4	$2.104 \cdot 10^6$	$1.979 \cdot 10^6$	$1.545 \cdot 10^6$
ZDT6	$2.752 \cdot 10^6$	$2.599 \cdot 10^6$	$1.965 \cdot 10^6$

По результатам тестирования можно понять, что при достаточно больших данных алгоритм построения выпуклых оболочек на всех слоях имеет выигрыш как по времени, так и по числу сравнений значений плотности заселения. А алгоритм, который строит оболочки только на последнем слое, не показал такого преимущества на практике. На маленьких тестах видно, что обычный линейный алгоритм работает лучше, из-за большой константы в сложности от геометрических преобразований и использования памяти, в алгоритмах построения выпуклых оболочек.

Таблица 3 – Время работы. Размер поколения 3000, число итераций $500\,000$

Problem	INDS	INDS LastHull	INDS AllHulls
DTLZ1	$1.319 \cdot 10^1$	$6.581 \cdot 10^0$	$6.229 \cdot 10^0$
DTLZ2	$1.749 \cdot 10^1$	$7.741 \cdot 10^0$	$7.511 \cdot 10^0$
DTLZ3	$5.646 \cdot 10^0$	$4.975 \cdot 10^0$	$5.156 \cdot 10^0$
DTLZ4	$2.371 \cdot 10^1$	$1.205 \cdot 10^1$	$1.120 \cdot 10^1$
DTLZ5	$2.514 \cdot 10^1$	$1.270 \cdot 10^1$	$1.140 \cdot 10^1$
DTLZ6	$3.414 \cdot 10^1$	$1.707 \cdot 10^1$	$1.586 \cdot 10^1$
DTLZ7	$2.641 \cdot 10^1$	$1.466 \cdot 10^1$	$1.384 \cdot 10^1$
WFG1	$2.691 \cdot 10^1$	$1.280 \cdot 10^1$	$1.198 \cdot 10^1$
WFG2	$6.634 \cdot 10^1$	$2.576 \cdot 10^1$	$2.063 \cdot 10^1$
WFG3	$3.753 \cdot 10^1$	$1.553 \cdot 10^1$	$1.367 \cdot 10^1$
WFG4	$3.682 \cdot 10^1$	$1.574 \cdot 10^1$	$1.425 \cdot 10^1$
WFG5	$5.933 \cdot 10^1$	$1.594 \cdot 10^1$	$1.314 \cdot 10^1$
WFG6	$1.754 \cdot 10^1$	$8.485 \cdot 10^0$	$7.361 \cdot 10^0$
WFG7	$2.787 \cdot 10^1$	$1.171 \cdot 10^1$	$9.837 \cdot 10^0$
WFG8	$2.883 \cdot 10^0$	$2.987 \cdot 10^0$	$3.137 \cdot 10^0$
WFG9	$2.093 \cdot 10^1$	$9.480 \cdot 10^0$	$8.597 \cdot 10^0$
ZDT1	$1.712 \cdot 10^1$	$9.380 \cdot 10^0$	$8.557 \cdot 10^0$
ZDT2	$1.622 \cdot 10^1$	$8.926 \cdot 10^0$	$8.427 \cdot 10^0$
ZDT3	$1.552 \cdot 10^1$	$8.860 \cdot 10^0$	$7.984 \cdot 10^0$
ZDT4	$1.084 \cdot 10^1$	$5.997 \cdot 10^0$	$5.845 \cdot 10^0$
ZDT6	$1.055 \cdot 10^1$	$7.833 \cdot 10^0$	$1.133 \cdot 10^1$

Таблица 4 – Число сравнений. Размер поколения 3000, число итераций $500\,000$

Problem	INDS	INDS LastHull	INDS AllHulls
DTLZ1	$8.999 \cdot 10^8$	$1.153 \cdot 10^8$	$1.224 \cdot 10^8$
DTLZ2	$1.236 \cdot 10^9$	$1.449 \cdot 10^8$	$1.566 \cdot 10^8$
DTLZ3	$3.259 \cdot 10^8$	$7.801 \cdot 10^7$	$8.095 \cdot 10^7$
DTLZ4	$1.173 \cdot 10^9$	$1.385 \cdot 10^8$	$1.498 \cdot 10^8$
DTLZ5	$1.236 \cdot 10^9$	$1.450 \cdot 10^8$	$1.563 \cdot 10^8$
DTLZ6	$1.721 \cdot 10^9$	$2.013 \cdot 10^8$	$1.994 \cdot 10^8$
DTLZ7	$1.244 \cdot 10^9$	$1.629 \cdot 10^8$	$1.788 \cdot 10^8$
WFG1	$1.287 \cdot 10^9$	$1.579 \cdot 10^8$	$1.539 \cdot 10^8$
WFG2	$3.444 \cdot 10^9$	$3.201 \cdot 10^8$	$3.252 \cdot 10^8$
WFG3	$1.833 \cdot 10^9$	$1.939 \cdot 10^8$	$2.049 \cdot 10^8$
WFG4	$1.908 \cdot 10^9$	$1.963 \cdot 10^8$	$2.057 \cdot 10^8$
WFG5	$3.078 \cdot 10^9$	$2.962 \cdot 10^8$	$3.125 \cdot 10^8$
WFG6	$1.304 \cdot 10^9$	$1.590 \cdot 10^8$	$1.608 \cdot 10^8$
WFG7	$2.131 \cdot 10^9$	$2.171 \cdot 10^8$	$2.260 \cdot 10^8$
WFG8	$8.908 \cdot 10^7$	$4.139 \cdot 10^7$	$4.230 \cdot 10^7$
WFG9	$1.641 \cdot 10^9$	$1.751 \cdot 10^8$	$1.995 \cdot 10^8$
ZDT1	$1.169 \cdot 10^9$	$1.614 \cdot 10^8$	$1.748 \cdot 10^8$
ZDT2	$1.076 \cdot 10^9$	$1.552 \cdot 10^8$	$1.665 \cdot 10^8$
ZDT3	$9.884 \cdot 10^8$	$1.451 \cdot 10^8$	$1.584 \cdot 10^8$
ZDT4	$6.735 \cdot 10^8$	$1.036 \cdot 10^8$	$1.081 \cdot 10^8$
ZDT6	$6.375 \cdot 10^8$	$1.283 \cdot 10^8$	$1.332 \cdot 10^8$

Таблица 5 – Время работы. Размер поколения $300\,000$, число итераций $20\,000\,000$

Problem	INDS	INDS AllHulls
ZDT1	$1.180 \cdot 10^4$	$2.639 \cdot 10^3$
ZDT2	$7.453 \cdot 10^3$	$2.583 \cdot 10^3$

Таблица 6 – Число сравнений. Размер поколения $300\,000$, число итераций $20\,000\,000$

Problem	INDS	INDS AllHulls
ZDT1	$2.597 \cdot 10^{11}$	$2.046 \cdot 10^{10}$
ZDT2	$1.682 \cdot 10^{11}$	$1.703 \cdot 10^{10}$

ЗАКЛЮЧЕНИЕ

В работе был предложен алгоритм оптимизации удаления из структуры данных INDS для инкрементальной недоминирующей сортировки с помощью выпуклых оболочек. Также был предложен метод построения выпуклых оболочек только для последнего слоя доминирования, без потери в асимптотической сложности времени работы алгоритма. Структура данных может быть использована в реализациях эволюционных алгоритмов для решения многокритериальных задач с большой популяцией особей.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 A Fast Elitist Multi-Objective Genetic Algorithm: NSGA-II / K. Deb [et al.] // Transactions on Evolutionary Computation. — 2000. — Vol. 6. — P. 182–197.

2 *Corne D. W.*, *Knowles J. D.*, *Oates M. J.* The Pareto Envelope-based Selection Algorithm for Multiobjective Optimization // Parallel Problem Solving from Nature Parallel Problem Solving from Nature VI. — Springer, 2000. — P. 839–848. — (Lecture Notes in Computer Science ; 1917).

3 PESA-II: Region-based Selection in Evolutionary Multiobjective Optimization / D. W. Corne [et al.] // Proceedings of Genetic and Evolutionary Computation Conference. — Morgan Kaufmann Publishers, 2001. — P. 283–290.

4 *Zitzler E.*, *Laumanns M.*, *Thiele L.* SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization // Proceedings of the EUROGEN'2001 Conference. — 2001. — P. 95–100.

5 *Knowles J. D.*, *Corne D. W.* Approximating the Nondominated Front Using the Pareto Archived Evolution Strategy // Evolutionary Computation. — 2000. — Vol. 8, no. 2. — P. 149–172.

6 *Abbass H. A.*, *Sarker R.*, *Newton C.* PDE: A Pareto Frontier Differential Evolution Approach for Multiobjective Optimization Problems // Proceedings of the Congress on Evolutionary Computation. — IEEE Press, 2001. — P. 971–978.

7 *Kung H. T.*, *Luccio F.*, *Preparata F. P.* On Finding the Maxima of a Set of Vectors // Journal of ACM. — 1975. — Vol. 22, no. 4. — P. 469–476.

8 *Jensen M. T.* Reducing the Run-time Complexity of Multiobjective EAs: The NSGA-II and Other Algorithms // Transactions on Evolutionary Computation. — 2003. — Vol. 7, no. 5. — P. 503–515.

9 *Fortin F.-A.*, *Grenier S.*, *Parizeau M.* Generalizing the Improved Run-time Complexity Algorithm for Non-dominated Sorting // Proceeding of Genetic and Evolutionary Computation Conference. — ACM, 2013. — P. 615–622.

10 *Buzdalov M.*, *Shalyto A.* A Provably Asymptotically Fast Version of the Generalized Jensen Algorithm for Non-Dominated Sorting // International Conference on Parallel Problem Solving from Nature. — 2014. — P. 528–537. — (Lecture Notes in Computer Science ; 8672).

11 *Nebro A. J.*, *Durillo J. J.* On the Effect of Applying a Steady-State Selection Scheme in the Multi-Objective Genetic Algorithm NSGA-II // Nature-Inspired Algorithms for Optimisation. — Springer Berlin Heidelberg, 2009. — P. 435–456. — (Studies in Computational Intelligence ; 193).

12 Efficient Non-domination Level Update Approach for Steady-State Evolutionary Multiobjective Optimization: tech. rep. / K. Li [et al.]. — 2014. — URL: `www.egr.msu.edu/~kdeb/papers/c2014014.pdf`.

13 *Buzdalov M.*, *Yakupov I.*, *Stankevich A.* Fast Implementation of the Steady-State NSGA-II Algorithm for Two Dimensions Based on Incremental Non-Dominated Sorting // Proceedings of Genetic and Evolutionary Computation Conference. — 2015 (to appear).

14 *Preparata F. P.*, *Shamos M. I.* Computational Geometry an Introduction. — Springer New York : 1985.