УДК 519.687.7
https://doi.org/10.36906/AP-2020/09

ОПТИМИЗАЦИЯ АЛГОРИТМА СОРТИРОВКИ TIMSORT

Захаров Д. М.
Нижневартовский государственный университет
г. Нижневартовск, Россия
Слива М. В.
канд. пед. наук
Нижневартовский государственный университет
г. Нижневартовск, Россия

Аннотация. В данной статье рассказывается о возможном варианте оптимизации алгоритма сортировки Timsort.

Ключевые слова. Сортировка, временная сложность, оптимизация, Timsort.

Одной из важных задач в информатике является нахождение оптимальных алгоритмов сортировок для разных входных данных. В 2002 году американский разработчик программного обеспечения Тим Петерс опубликовал устойчивый, адаптивный нерекурсивный алгоритм сортировки под названием Timsort (https://clck.ru/T7TNs). Данный алгоритм умело совмещает сортировку вставками и сортировку слиянием, добиваясь высокого быстродействия на так называемых «реальных» данных, представляющих из себя массивы, состоящие из упорядоченных частей.

Timsort имеет следующие характеристики алгоритмической сложности:

Худшее время — $O(n \log(n))$;

Среднее время — $O(n \log(n))$;

Лучшее время — $O(n)$.

Вкратце процесс сортировки состоит из следующих шагов:

1. Вычислить значение для константы «minrun», определяющей минимальный размер упорядоченной части. Обычно значение находится в диапазоне (32; 65).

2. Определить границы следующего отсортированного участка. Если он отсортирован по убыванию, обратить порядок его элементов. Если длина подмассива меньше константы «minrun», добавить недостающие элементы бинарными вставками.

3. Произвести слияние подмассива с предыдущим по определенным правилам (https://clck.ru/T7TLj).

Одна из сильных сторон данного алгоритма состоит в том, что на массивах, отсортированных по убыванию и без одинаковых элементов, временная сложность становится линейной, так как происходит всего одна операция обращения массива. Однако, если обратный массив содержит неуникальные элементы, сложность возвращается к линейно-логарифмической. Более того, к таким данным часто применяется сортировка вставками, для которой обратно отсортированный массив является худшим случаем и работает за квадратичное время.

Таким образом, чтобы сохранить линейное время на обратно отсортированных данных и не нарушить устойчивость алгоритма (не менять местами одинаковые элементы), необходимо определить функцию устойчивого обращения элементов.

Рассмотрим простейший вариант реализации данной функции, написанный на языке Lua для собственной реализации сортировки.

```
local function reverse(left, right)
    local n = left + right

    for i = left, ceil(n / 2) - 1 do
        list[i], list[n - i] =
        list[n - i], list[i]
    end
end

local function stablereverse(left, right)
    -- произвести обычное обращение
    reverse(left, right)

    local i = left

    -- определить верхнюю границу подмассива с одинаковыми элементами
    while i < right do
        local item = list[i]
        local j = i + 1

        while j <= right and not cmp(item, list[j]) do
            j = j + 1
        end

        -- обратить порядок одинаковых элементов
        reverse(i, j - 1)
        i = j
    end
end
```

Процедура обычного обращения элементов определена под именем «reverse». Функция сравнения, передаваемая алгоритму сортировки, имеет имя «cmp». Она принимает 2 сортируемых элемента и возвращает «true», если первый меньше второго, иначе «false». Функция «stablereverse» сначала применяет процедуру «reverse» к убывающим элементам на заданном отрезке, затем определяет границы частей с одинаковыми элементами и обращает их, восстанавливая исходный порядок элементов.

Цена данной оптимизации состоит в необходимости добавления дополнительных сравнений при определении границ упорядоченных частей.

Для оценки эффективности оптимизации была написана программа, выводящая график зависимости времени работы алгоритма от количества входных данных.

Горизонтальная линия отражает размер массива, а вертикальная – время сортировки в миллисекундах.

Предположим, что средний случай для оптимизации состоит в том, что устойчивые обращения выполняются приблизительно в половине случаев. То есть входной массив состоит из упорядоченных по убыванию и по возрастанию частей с неуникальными элементами.

Результат работы программы представлен на рисунке 1.

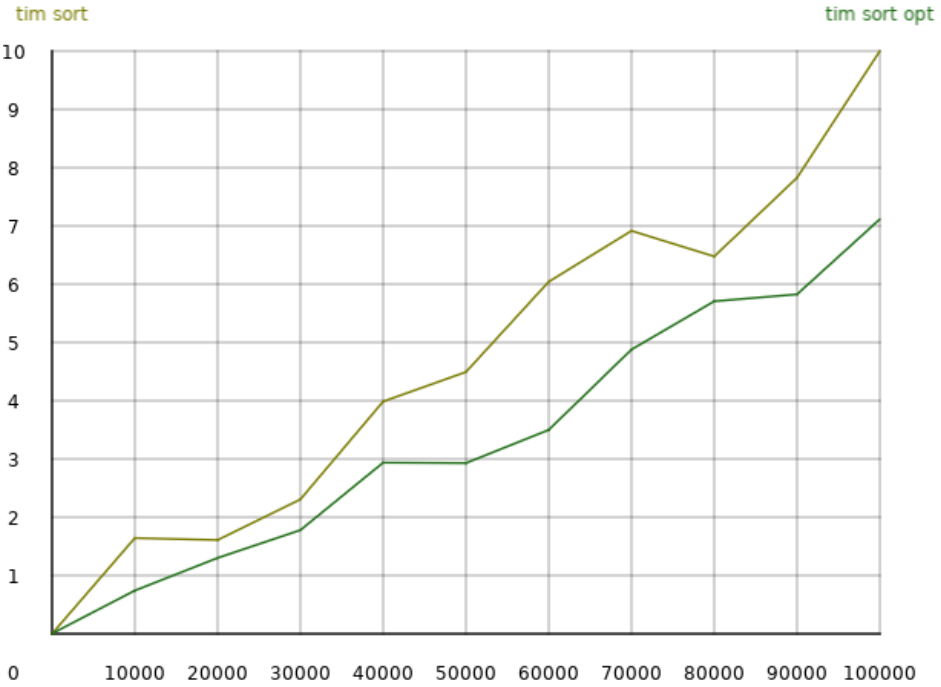

Рис. 1. График времени работы алгоритмов на «реальных» данных

Желтая линия отражает результаты классической реализации Timsort, зеленая – оптимизированной реализации. Согласно полученным данным, оптимизированный вариант в среднем на 12% быстрее обычного.

Теперь рассмотрим худший случай оптимизации – хаотичный массив без уникальных элементов. На таких данных устойчивое обращение никогда не выполняется, и добавленные дополнительные сравнения являются лишними. Результат работы программы представлен на рисунке 2.

Как видно из графика, разница во времени работы сортировок на любых размерах входных данных пренебрежимо мала, что говорит о низкой стоимости оптимизации.

Таким образом, данная модификация алгоритма сортировки Timsort позволяет добиться более высокого быстродействия на «реальных» данных, не ухудшая производительность на хаотичных массивах.

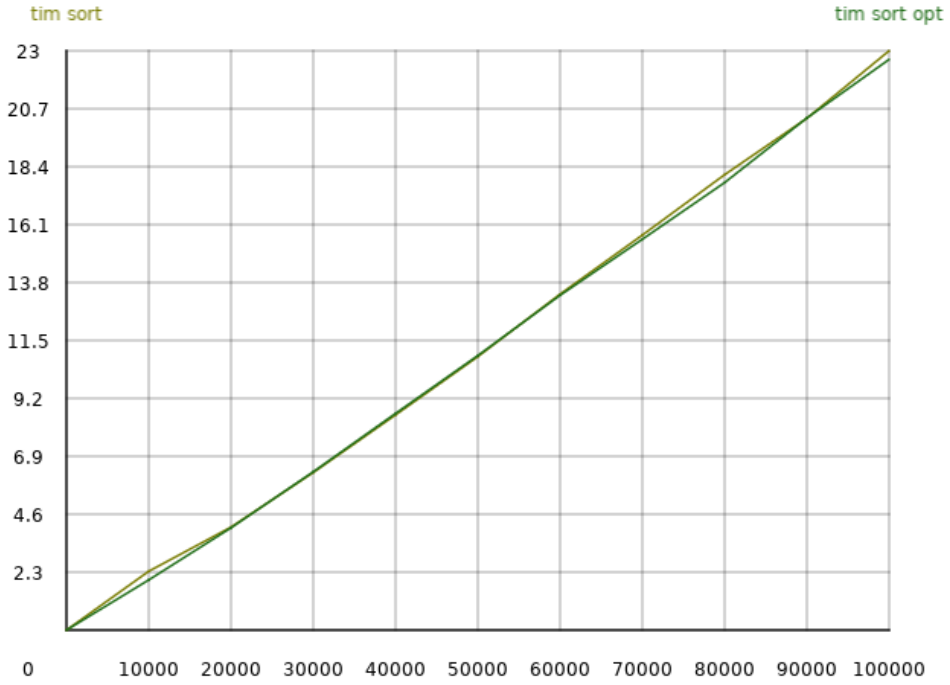

Рис. 2. График времени работы алгоритмов на хаотичных данных