Applied Mathematics & Information Sciences
An International Journal

On the Worst Case Data Sets for Order Statistics

Lei Wang[1] *and Xiaodong Wang*[2,3,*]

[1] Microsoft AdCenter, Bellevue, WA 98004, USA
[2] School of Mathematics and Computer Science, Fuzhou University, 350002 Fuzhou , China
[3] School of Computer Science, Quanzhou Normal University, 362000 Quanzhou , China

Received: Jul 8, 2011; Revised Oct. 4, 2011; Accepted Dec. 26, 2011
Published online: 1 May 2012

Abstract: In this work, we consider the selection algorithms for the order statistics problems. A general partition based selection algorithm can be made to go quadratic by constructing input on the fly in response to the sequence of items compared. We develop an extremely simple class for constructing the worst case data set for the partition based selection algorithm. The general method works against any implementation of partition based selection algorithm that satisfies certain very mild and realistic assumptions. Computational results ascertain that the techniques developed are not only of theoretical interest, but also may actually lead to the worst case data sets for general partition based selection algorithms.

Keywords: Order statistics, Adversary, Worst Case, Algorithms.

1. Introduction

In this paper, we consider the selection algorithms for the order statistics problems. The kth order statistic of a statistical sample is equivalent to its kth-smallest value. It is the most fundamental tools in non-parametric statistics and inference. In computer science, an algorithm for solving the order statistics problems is also called a selection algorithm[2,4]. The task of a selection algorithm is to find the kth smallest item in a set. This item is called the kth order statistic. This task includes the cases of finding the minimum, maximum, and median items. There are $O(n)$, worst-case linear time selection algorithms[3,5]. Selection is also a subproblem of more complex problems like the nearest neighbor problem and shortest path problems.

A general partition based selection algorithm is known as Hoare's selection algorithm or quickselect[6]. In the algorithm quickselect, there is a sub-algorithm called partition that can, in linear time, group a set, ranging from indices left to right, into two parts, those less than a certain item, and those greater than or equal to the item. Although quickselect is linear-time on average and therefore efficient in practice, it can require quadratic time with poor pivot choices.

When using quickselect we may feel a nagging tension that it will go quadratic. Tactics to avoid embarrassing results in some low-entropy cases, such as already ordered input, are not difficult to discover. Nevertheless, production implementations have been caught going quadratic in real-life applications. No matter how hard an algorithm tries, it cannot defend against all inputs without great sacrifice of its speed.

This paper describes an adversarial method that finds chinks in the defenses of any implementation. A similar method was presented by Mcilroy for quicksort algorithm[7], but our method is somewhat different. A polymorphic implementation of quickselect never looks at the data. It relies instead on an externally supplied comparison function. This allows us to monitor and influence the algorithm non-invasively. For this purpose, we make a comparison function that observes the pattern of comparisons and constructs adverse data on the fly.

The organization of the paper is as follows.

In the following 3 sections we describe our general adversary class design paradigm. In section 2 we give an extremely simple class for constructing the worst case data set for the partition based selection algorithm. The general method works against any implementation of partition based selection algorithm that satisfies some mild and realistic assumptions. In section 3 we give a computational study of the presented adversary class which demonstrates that the achieved results are not only of theoretical interest,

* Corresponding author: e-mail: wangxiaodong@qztc.edu.cn

but also that the techniques developed may actually lead to the worst case data sets for general partition based selection algorithms. Some concluding remarks are in section 4.

2. The Adversary Design

Any partition based selection algorithm finds the kth smallest item in a set of n data items by using the so-called divide and conquer strategy in three phases:

1. Pick a data item as pivot. This operation costs $O(1)$ comparisons.

2. Partition the data into three parts that respectively contain all j items less than the pivot, the pivot item itself, and all items greater than the pivot. The placement of items equal to the pivot varies among implementations.

3. The algorithm recursively continues in the appropriate part of data: the low part if $k < j$, high part if $k > j$.

For this kind of partition based selection algorithms, we will construct an adversary to make the selection algorithms go quadratic by arranging for the pivot to be one of the end points of the interval containing all items not seen during pivot selection so that the partition will be lopsided.

Those items whose relationship to each other is unknown are considered to have a value of gas[7]. The exact values are not determined as long as they are not compared against each other. Quadratic behavior is guaranteed since $n - O(1)$ gas values must survive pivot selection among n items.

In the whole of the algorithm, all n items are divided into three parts, the lower part, the gas part and the upper part. The values of the items in gas part are not determined. All the items of the lower part and the upper part have solid values. The value of an item of the lower part is less than the value of any item of the upper part. In the initial part of the algorithm, all items are gas items. The lower part and the upper part of the items are both empty.

When two gas items are compared, one gets frozen into a solid lower part item or a solid upper part item according to the value of k.

When a lower part item is compared to a gas item, it compares low. When an upper part item is compared to a gas item, it compares high. When two solid items are compared, the answer depends on the frozen values.

The trick of the adversary is to make sure that the pivot gets frozen early in the partition phase if it has not already been frozen. No further gas items will become frozen for the duration of the partitioning phase.

A simple observation helps to guess the pivot and freeze it. A pivot candidate is the gas item that most recently survived a comparison. When an item is to be frozen in a gas-gas comparison, a pivot candidate is preferred. The pivot candidate will emerge as soon as a gas item is examined. If the pivot is already solid, the candidate is not important. Otherwise, the first gas-gas comparison in the partition phase results in the pivot either getting frozen or becoming the pivot candidate. The pivot will become frozen at the second gas-gas comparison of the partitioning phase in the worst case. At most two items will be frozen during partitioning. This assures that $n - O(1)$ items will be survival.

The adversarial method works for almost any polymorphic program recognizable as a partition based selection algorithm. The algorithm may copy values at will, or work with lists rather than arrays. It may even pick the pivot at random. The partition based selection algorithm must satisfy some mild assumptions:

1. Pivot-choosing takes $O(1)$ comparisons.

2. The comparisons of the partitioning phase are contiguous and involve the pivot value.

3. The only data operations performed are comparison and copying.

Based on the discussion above, we can design an adversary class for constructing the worst case data items for a given partition based selection algorithm as follows.

```
template <class T>
class adversary
{ // the k_th smallest item
  static T k;
  // gas value
  static T gas;
  // the largest value of lower part
  static T lower;
  // the smallest value of upper part
  static T upper;
  // pivot candidate
  static T * candidate;
  // number of comparisons
  static long long ncmp;
  // item values
  mutable T * val;
  void freeze() const{
     *val=lower<k ?
        T(lower++):T(upper--);
  }
public:
  static std::vector<T *> memory;
  adversary() : val(0) {}
  void init() {
    memory.push_back(val = new T(gas));
  }
  void nset(T kth, T n){
    k=kth;
    gas=n;
    upper=n+n;
  }
  bool operator<
    (const adversary & other) const
};
```

In the description of the adversary class above, the variable k is the parameter of the selection algorithm to find the kth smallest of the data set. The variable $lower$ stores the largest item value of the current lower part and the vari-

able $upper$ stores the smallest item value of the current upper part. The variable gas is for the gas value. Gas is coded as a middle value. It is larger than any item value of the lower part and less than any item value of the upper part. The variable $candidate$ is for the pivot candidate. When we have to freeze an item in a gas-gas comparison, the pivot candidate is chosen. The initial values of gas and the upper value are dependent on the size of the data set. If the size of the data set is n, then the function $nset$ will set the value of gas to n and the initial value of $upper$ to $2n$.

The variable val is a pointer pointing to the frozen item values. The vector $memory$ is the actual memory to store the adversarial data set constructed. It is public to allow recovery of the adversarial data set after the selection algorithm finished and to free the memory used.

A default copy constructor and assignment operator which just copy the val pointer is used. A default destructor is also used, which means the memory for the values will leak until we call collect.

The variable $ncmp$ is used to record the number of comparisons used in the current state of the selection algorithm.

The most important operation of the adversary class is the comparison operation '<'.

```
bool operator<(const adversary & other)
  const
{
  ncmp++;
  if (*val == gas && *other.val == gas)
  {
      if (val == candidate)freeze();
      else other.freeze();
  }
  if (*val == gas) candidate = val;
  else if (*other.val == gas)
     candidate = other.val;
  return *val < *other.val;
}
```

Since the pivot candidate is the gas item that most recently survived a comparison, when the pivot candidate involved in a gas-gas comparison, it must be frozen by the function $freeze$. The function $freeze$ freezes the pivot candidate to a lower part item or an upper part item according to the value of k to ensure that the kth smallest item is neither in the lower part nor in the upper part of the current data set.

The other comparison operations can be transformed into comparison operation $<$.

```
  bool operator>(const adversary & a)
  {return a.operator<(*this);}

  bool operator<=(const adversary & a)
  {return !(operator>(a));}

  bool operator>=(const adversary & a)
  {return !(operator<(a));}

  bool operator==(const adversary & a)
  {return val == a.val;}

  bool operator!=(const adversary & a)
  {return val != a.val;}
```

By using this adversary class, we can construct the worst case data set for the partition based selection algorithm satisfying the mild assumptions mentioned above.

Suppose we want to construct the worst case data set for the algorithm $select$ and we define sequence_type as follows.

```
#define set_type vector
typedef adversary<unsigned> U;
typedef set_type<U>::iterator iter;
```

The algorithm $construct$ for this purpose can be described as follows.

```
void construct(int n,float k)
{
    set_type<U> seq(n);
    first=seq.begin();
    last=seq.end();
    kth=first+(int)(k*(seq.size()-1));
    unsigned nn=last-first;
    unsigned kk=nth-first;
    seq.begin()->nset(kk,nn);
    for (iter i=seq.begin();
      i!=seq.end();++i)i->init();
    select(first,kth,last);
    output(U::ncmp,U::memory);
    release(U::memory);
}
```

In the algorithm $construct(n,k)$, the function $output$ is used to output the worst case data set for the algorithm $select$ and the number of comparisons used for the algorithm $select$ on this data set. The function $release$ is used to release the memory allocated for the data set.

3. Computational Experiments

In this section, we performed a series of experiments on various partition based selection algorithms. The investigated 9 algorithms are listed in Table 1.

For these investigated algorithms we construct their worst case data sets by using the adversary class described above with various sizes of the data sets. For example, we construct the worst case data set for the STL implementation of selection algorithm **nth_element** of size 100 as shown in Table 2. For this data set, the algorithm **nth_element** needs 3587 comparisons to find the median.

Table 1 Selection algorithms investigated

Algorithm	Description
select	A partition based selection algorithm which chooses the middle item of the data set as the pivot[4].
rselect	A partition based selection algorithm which chooses a random item of the data set as the pivot[4].
nth_element	STL implementation of selection algorithm which chooses the median of three items first, middle and last items of the data set as the pivot[1].
select-med3t	Tukey's ninther pivot selecting algorithm which chooses the median of the medians of three items, each of three random items of the data set as the pivot[1].
select-medt5	A median-of-$(2t+1)$ pivot selecting algorithm with $t=5$ which chooses the median of 11 random items of the data set as the pivot[1].
select-medt9	A median-of-$(2t+1)$ pivot selecting algorithm with $t=9$ which chooses the median of 19 random items of the data set as the pivot[1].
iselect	Selection algorithm using $2x+b$ rule introspection but no randomization[8].
riselect	Selection algorithm using $(6/5)x+b$ rule introspection and randomization[10]
linear	Blum's worst case linear time selection algorithm[3].

Table 2 The worst case data set for **nth_element** of size 100

1	2	3	100	151	5	152	7	153
9	154	11	155	13	156	15	157	17
158	19	159	21	160	23	161	25	162
27	163	29	164	31	165	33	166	35
167	37	168	39	169	41	170	43	171
45	172	47	173	200	0	4	6	8
10	12	14	16	18	20	22	24	26
28	30	32	34	36	38	40	42	44
46	48	174	175	176	177	178	179	180
181	182	183	184	185	186	187	188	189
190	191	192	193	194	195	196	197	198
199								

The 9 algorithms experimented are divided into three groups. For the algorithms in each group we construct the worst case data sets by using our adversary class with the data sizes in the range of $10 \leq n \leq 300000$. The algorithms in groups 1 and 2 are all partition based selection algorithms. The algorithms in groups 2 are of more subtle pivot choices. The algorithms in groups 3 are not partition based selection algorithms. They are worst case linear time algorithms actually.

Table 3 The number of comparisons for algorithms in group 1

n	select	rselect	nth_element
10	57	64	74
50	927	898	815
100	3340	3338	3587
300	28790	28787	29537
500	79240	79236	80487
700	154690	154687	156437
900	255140	255133	257387
1000	314740	314729	317237
3000	2819240	2823677	2826737
5000	7823740	7823734	7836237
7000	15328240	15331766	15345737
9000	25332740	25337756	25355237
10000	31272490	31278511	31297487
30000	281317490	281317481	281392487
50000	781362490	781362488	781487487
70000	1531407490	1531407481	1531582487
90000	2531452490	2531452479	2531677487
100000	3125224990	3125224980	3125474987
150000	7031587490	7031587481	7031962487
200000	12500449990	12500554130	12500949987
250000	19531812490	19532178413	19532437487
300000	28125674990	28126018567	28126424987

For each worst case data set constructed we count the number of comparisons required by the corresponding selection algorithms.

The experiment results for the three groups are reported in Table 3-5. The results reported here were obtained on a personal computer with Pentium(R) Dual Core CPU 2.10 GHz and 2.0 Gb RAM, using the Microsoft Visual C++ version 8.0 compilers. The word size of the processor is $w = 32$.

The experiment results show that the techniques suggested in this paper produce real worst case data sets for all partition based selection algorithms. For the results reported above, we estimated the time complexities for each algorithm performed on the corresponding data set by using a SAS regression program.

The adversary is effective, as Table 6 shows. For the algorithms in group 1, the coefficients of the quadratic items of their time complexities are 0.3125. For the algorithms in group 2, the coefficients of the quadratic items of their time complexities are all greater than 0.0466. This means that the time complexities of the algorithms in these two groups are really quadratic for the worst case data sets constructed by our adversary class. Our adversary class does not work for the algorithms in group 3, since they are all worst case linear time algorithms actually.

4. Concluding Remarks

We have suggested a novel technique for constructing the worst case data sets for general partition based selection al-

Table 4 The number of comparisons for algorithms in group 2

n	**select-med3t**	**select-medt5**	**select-medt9**
10	71	57	57
50	782	714	772
100	1509	1664	1861
300	9894	9358	8065
500	25683	23360	18087
700	50593	42542	31430
900	82463	68568	49282
1000	101108	82833	59114
3000	852074	695992	457134
5000	2458400	1903355	1226492
7000	4725897	3708364	2368485
9000	7939989	6110085	3883439
10000	9799473	7530145	4783367
30000	87991183	67247321	42280047
50000	244305266	186440553	117056979
70000	478747102	365255992	229081829
90000	791312369	603537970	378409316
100000	976912970	744936353	467081515
150000	2197765519	1675392905	1049982704
200000	3906957129	2977909154	1865948685
250000	6104344980	4652450331	2914653789
300000	8790049364	6698965999	4196492116

Table 5 The number of comparisons for algorithms in group 3

n	**iselect**	**riselect**	**linear**
10	74	76	74
50	815	840	815
100	921	2880	553
300	3665	8746	2117
500	5901	14871	3726
700	8735	21889	5142
900	11472	28280	6737
1000	12698	31824	7752
3000	40257	99733	25308
5000	67143	167257	42156
7000	94664	235283	60089
9000	121910	302445	77175
10000	135706	335485	86062
30000	414276	1012533	264679
50000	692855	1691997	443114
70000	971940	2372138	621598
90000	1252335	3050536	803597
100000	1392975	3392025	894278
150000	2093603	5090631	1344105
200000	2797588	6792541	1789645
250000	3500777	8497393	2248174
300000	4199384	10197364	2700286

Table 6 Worst case performance for various selection algorithms investigated

Algorithm	**Pivot choice**	**Number of comparisons**
select	middle item	$0.3125n^2 + 2.25n - 6$
rselect	random item	$0.3125n^2 + 1.6n + 3684$
nth_element	median of three items	$0.3125n^2 + 4.75n - 27$
select-med3t	Tukey's ninther	$0.0976n^2 + 3.6n - 10022$
select-medt5	median-of-$(2t+1)$ with $t = 5$	$0.0744n^2 + 9.21n + 1021$
select-medt9	median-of-$(2t+1)$ with $t = 9$	$0.0466n^2 + 11.74n + 787$
iselect	$2x+b$ introspection	$13.9n - 1746$
riselect	$(6/5)x + b$ introspection	$33.89n - 2061$
linear	worst case linear	$8.9n - 1378$

gorithms. The data sets constructed by our technique s indeed the worst case data sets for the algorithms in groups 1 and 2, which make the corresponding algorithms go quadratic. In fact, our technique will work for all partition based selection algorithms.

In this work we study the selection algorithms for the common order statistics problem where we want to find the kth-smallest item of a data set consisting of n items coming from a ordered complete set S.

In some cases, we know that the data set consists of n items coming from an ordered complete set $f(S) = \{f(X)|X \subseteq S\}$ where $f(X)$ is a function of X and may not be computed in constant time. For these cases, some partition based selection algorithms also work[9]. It is not clear that whether our presented technique can be used to construct the worst case data set for these algorithms. We will investigate the problem further.

Acknowledgement

The authors acknowledge the financial support of Science and Technology of Fengze under Grant No.2009FZ24 and the Haixi Project of Fujian under Grant No.A099. The authors are grateful to the anonymous referee for a careful checking of the details and for helpful comments that improved this paper.

References

[1] JL Bentley, MD McIlroy, Engineering a sort function. SoftwareâĂŤPractice and Experience 23(11), 1249-1265, 1993.
[2] R. Bird, Pearls of Functional Algorithm Design, Cambridge University Press, New York, 2010.
[3] M. Blum, RW Floyd, V. Pratt, RL Rivest, RE Tarjan, Time bounds for selection. Journal of Computer and System Sciences, 7(4), 448-461, 1973.

[4] T.H.Cormen, C.E.Leiserson, R.L.Rivest: *Introduction to Algorithms*, 2nd ed., MIT Press, Cambridge, 1990.
[5] Yevgeniy Dodis, Mihai Patrascu, Mikkel Thorup, Changing Base Without Losing Space, In *Proc. 42st ACM Symposium on Theory of Computing (STOC)*, 2010.
[6] CAR Hoare, Quicksort. Computer Journal 5, 10-15, 1962.
[7] MD McIlroy, A killer adversary for quicksort. Software‐Practice and Experience 29(4), 341-344, 1999.
[8] DR Musser, Introspective sorting and selection algorithms. Software‐Practice and Experience, 27(8), 983-993, 1997.
[9] A. Rauh and G. R. Arce, A Fast Weighted Median Algorithm Based on Quickselect, Proceedings of 2010 IEEE 17th International Conference on Image Processing, 105-108
[10] JD Valois, Introspective sorting and selection revisited, Software‐Practice and Experience, 30(4), 617-638, 2000.

Lei Wang, PhD in Computer Science from Georgia Institute of Technology 2011. Applied researcher at Microsoft. Has experience in computer science with emphasis in algorithm design. The areas of interest are approximation and randomized algorithms,mechanism design, market equilibrium computation.

Xiaodong Wang, Professor in Computer Science Department of Quanzhou Normal University and Fuzhou University,China. Has experience in computer science and applied mathematics. The areas of interest are design and analysis of algorithms,exponential-time algorithms for NP-hard problems,strategy game programming.