Report of experiments

O. Denas

12/28/2016

Contents

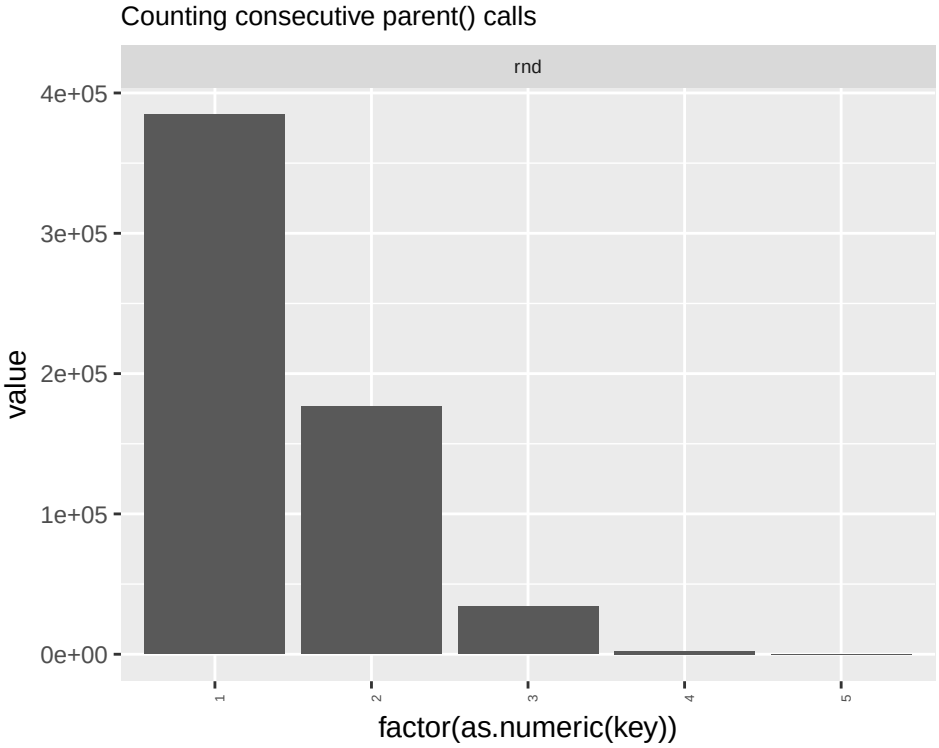
1 Input properties

For various types of inputs ("mut_XMs_YMt_Z" means `s` and `t` are random identical strings of length X, and Y million respectively with mutations inserted every Z characters. "rnd_XMs_YMt" means `s` and `t` are random strings of length X, and Y million respectively) run the MS algorithm and count the number of

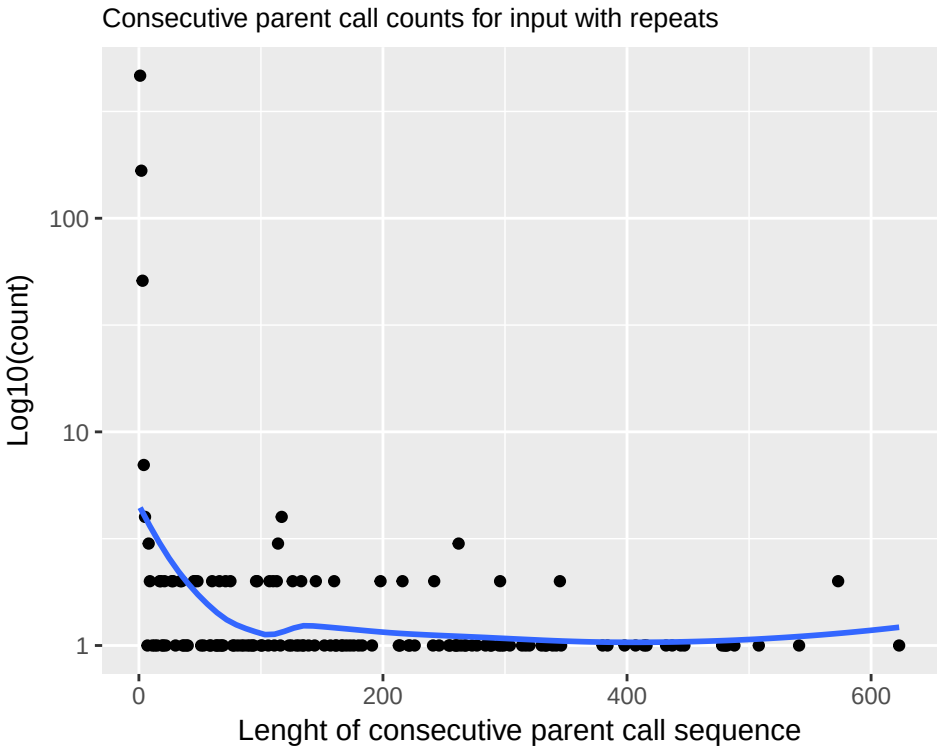
- consecutive `parent()` calls during the `runs` construction.
- consecutive `wl()` calls during the `ms` construction.
- the number of 1s in the `runs` bit vector
- double rank calls that fail (i.e the search down the WT is interrupted prior to reaching a leaf)
- the number of maximal repeats

1.1 Consecutive parent calls (RUNS construction)

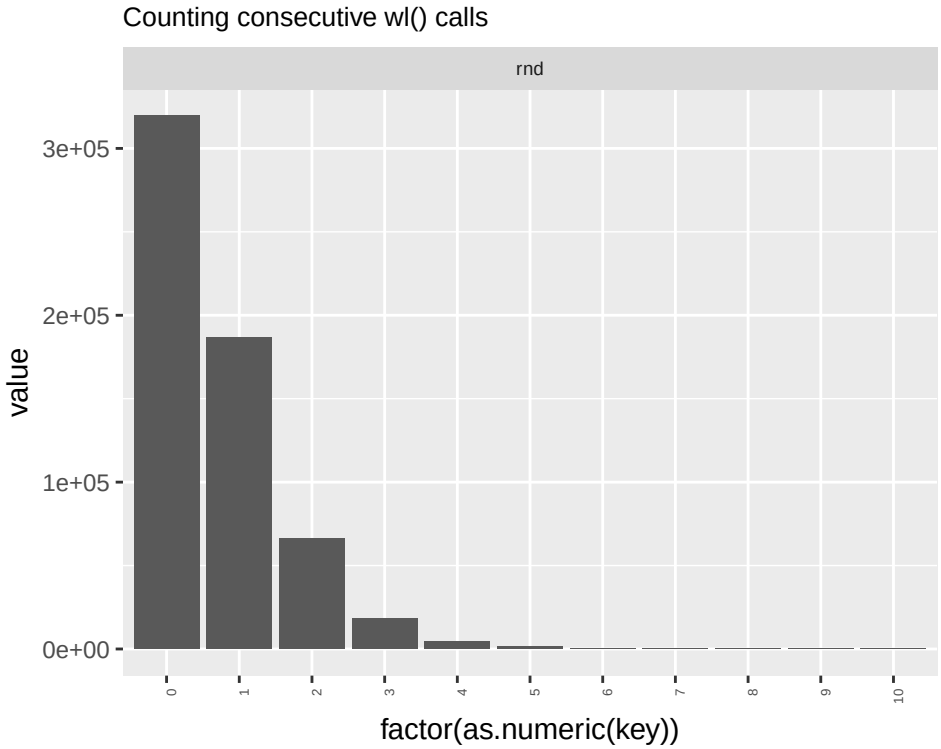
```
## Warning: Too few values at 617 locations: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
## 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...
```

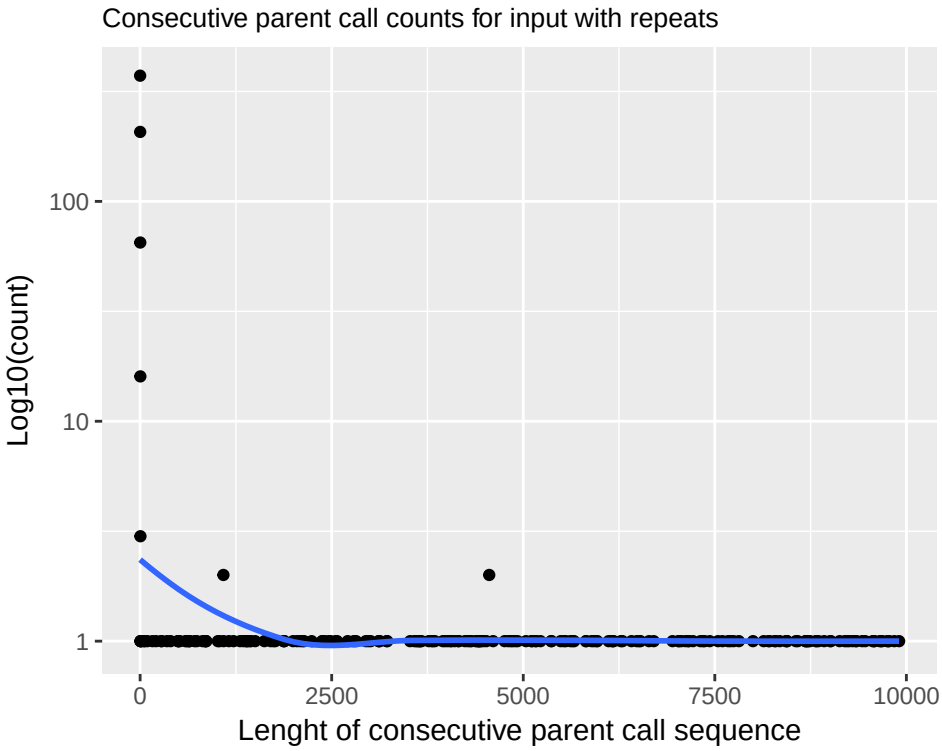


The input with repeats has a very different distribution from above.



1.2 Consecutive wl calls (MS construction)




Consecutive parent call counts for input with repeats
Log10(count)
100
10
1
0
2500
5000
7500
10000
Lenght of consecutive parent call sequence

1.3 Other stats

Table 1: Interval width for various input types.

inp_type	small	large	small_perc
rep	0	0	NaN
rnd	0	0	NaN
rep	0	0	NaN
rnd	0	0	NaN

Table 2: Distribution of node types (in percentage) when wl() calls are made.

maximality	wl_presence	iwidth	rep	rnd
maxrep	nowl	narrow	0.19	24.11
maxrep	nowl	wide	1.46	0.07
maxrep	wl	narrow	0.25	48.86
maxrep	wl	wide	1.38	0.05
nonmaxrep	nowl	narrow	7.95	6.70
nonmaxrep	nowl	wide	16.31	0.01
nonmaxrep	wl	narrow	23.70	20.19
nonmaxrep	wl	wide	48.75	0.01

2 Current performance

Table 3: Run time in seconds, on random input with |s| = 1MB, |t| = 5MB

lazy	fail	maxrep	total_s
1	1	1	92.992
0	1	0	93.066
1	1	0	93.252
0	1	1	93.822
0	0	1	94.613
0	0	0	94.935
1	0	1	94.956
1	0	0	95.083

3 Double vs. single rank

3.1 Rank support optimization

The optimization occurs first at `rank_support_v.hpp` where we avoid recomputing a major block for intervals that are going to fall on the same major block anyways.

The condition that checks whether endpoints (i, j) of an interval end up in the same major block is

```
bool((i>>8) == (j>>8))
```

3.1.1 Code

The single rank and double rank implementations in sdsl: rank_support_v.hpp link

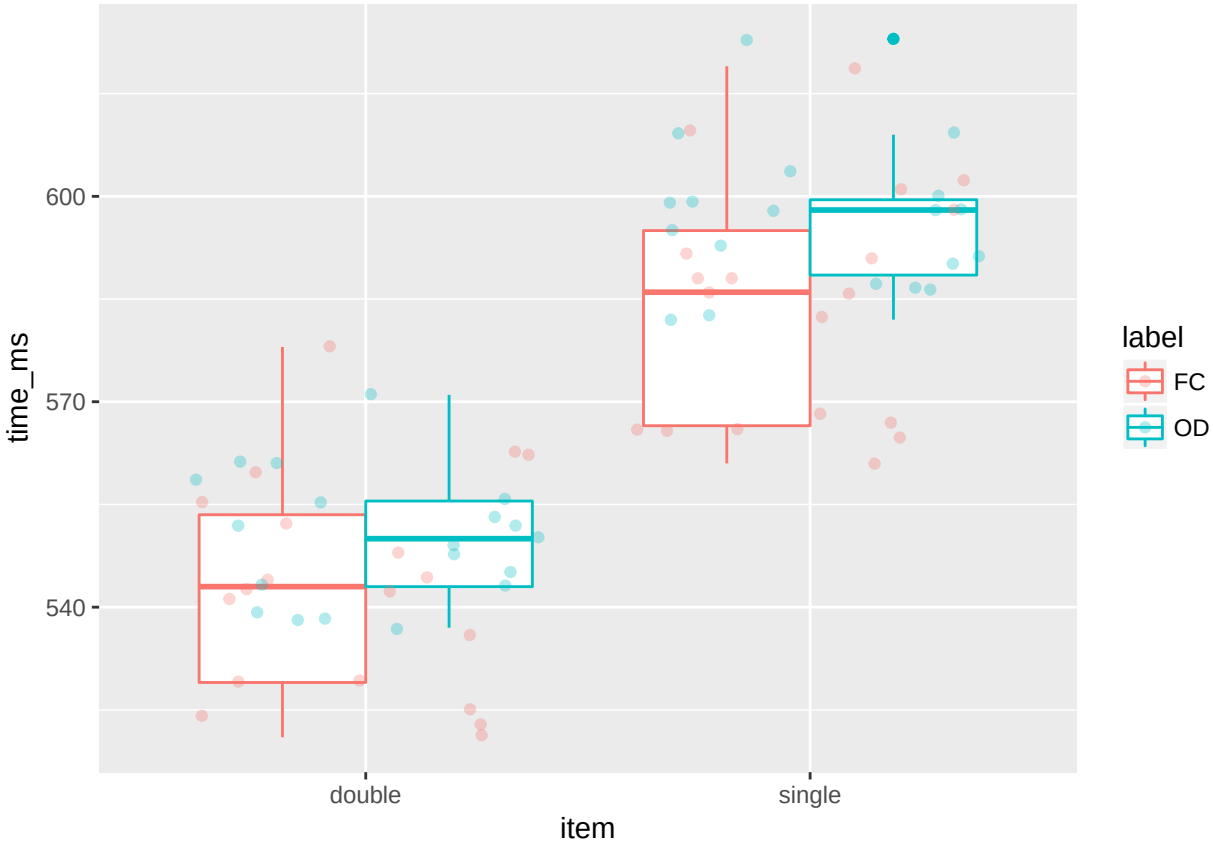
```
// RANK(idx)
const uint64_t* p = m_basic_block.data() + ((idx>>8)&0xFFFFFFFFFFFFFFFEULL);
return *p + ((*(p+1)>>(63 - 9*((idx&0x1FF)>>6)))&0x1FF) +
      (idx&0x3F ? trait_type::word_rank(m_v->data(), idx) : 0);

// DOUBLE RANK OD(i, j)
if((i>>8) == (j>>8)){
  const uint64_t* p = m_basic_block.data() + ((i>>8)&0xFFFFFFFFFFFFFFFEULL);
  res.first  = *p + ((*(p+1)>>(63 - 9*((i&0x1FF)>>6)))&0x1FF) +
            (i&0x3F ? trait_type::word_rank(m_v->data(), i) : 0);
  res.second = *p + ((*(p+1)>>(63 - 9*((j&0x1FF)>>6)))&0x1FF) +
            (j&0x3F ? trait_type::word_rank(m_v->data(), j) : 0);
} else {
  const uint64_t* p = m_basic_block.data() + ((i>>8)&0xFFFFFFFFFFFFFFFEULL);
  res.first  = *p + ((*(p+1)>>(63 - 9*((i&0x1FF)>>6)))&0x1FF) +
            (i&0x3F ? trait_type::word_rank(m_v->data(), i) : 0);
  p -= (((i>>8)&0xFFFFFFFFFFFFFFFEULL) - ((j>>8)&0xFFFFFFFFFFFFFFFEULL));
  res.second = *p + ((*(p+1)>>(63 - 9*((j&0x1FF)>>6)))&0x1FF) +
            (j&0x3F ? trait_type::word_rank(m_v->data(), j) : 0);
}
return res

// DOUBLE RANK FC(i, j)
const uint64_t* b = m_basic_block.data();
const uint64_t* pi = b + ((i>>8)&0xFFFFFFFFFFFFFFFEULL);
const uint64_t* pj = b + ((j>>8)&0xFFFFFFFFFFFFFFFEULL);

return (*pi + ((*(pi+1)>>(63 - 9*((i&0x1FF)>>6)))&0x1FF) +
             (i&0x3F ? trait_type::word_rank(m_v->data(), i) : 0),
      *pj + ((*(pj+1)>>(63 - 9*((j&0x1FF)>>6)))&0x1FF) +
             (j&0x3F ? trait_type::word_rank(m_v->data(), j) : 0));
```

3.1.2 Performance

The FC implementation seems to work better and will be adopted from now on.

Table 4: Time (in ms) of 500K calls to `wl()` based on `single_rank()` or `double_rank()` methods on 100MB random DNA input; Mean/sd over 20 repetitions.

item	label	avg_time	sd_time
double	FC	543.11	15.88
double	OD	550.00	9.27
single	FC	584.32	17.11
single	OD	596.37	10.20



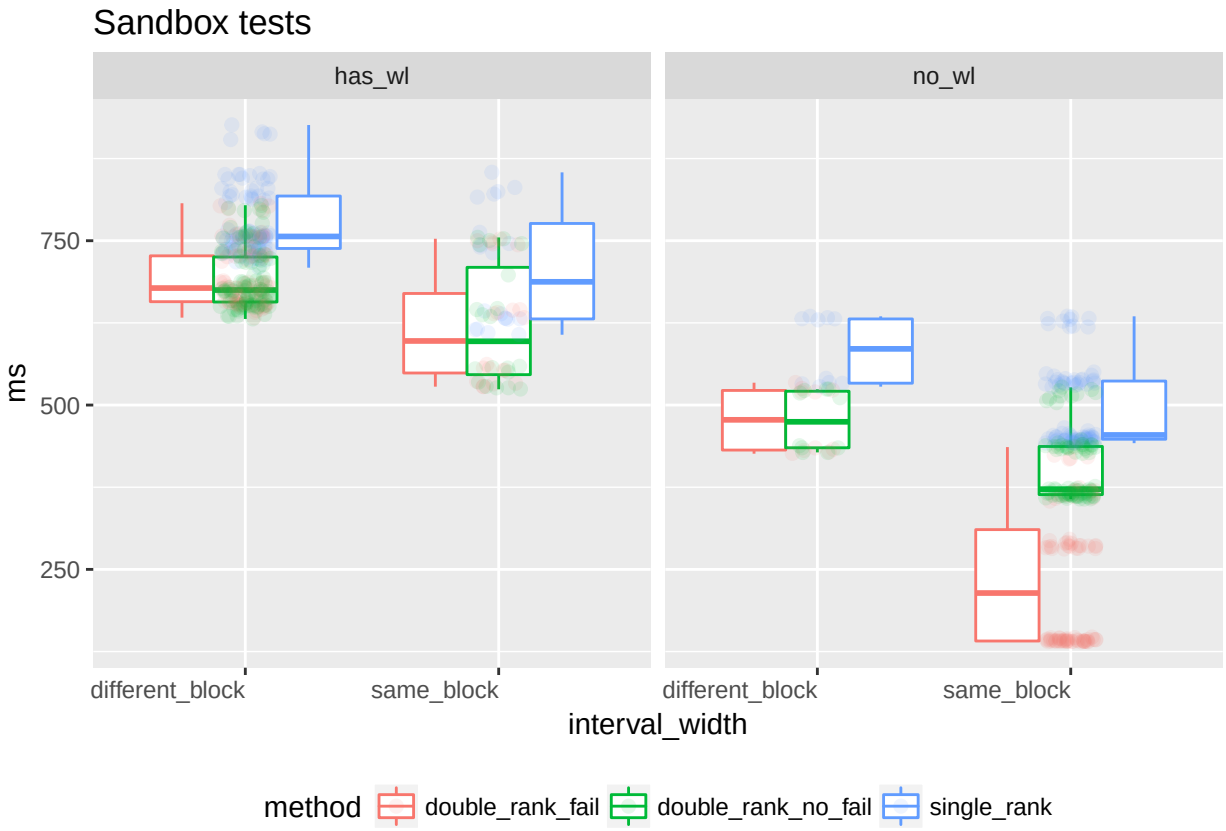
3.2 Weiner Link optimization – single vs. double rank

3.2.1 Sandbox performance

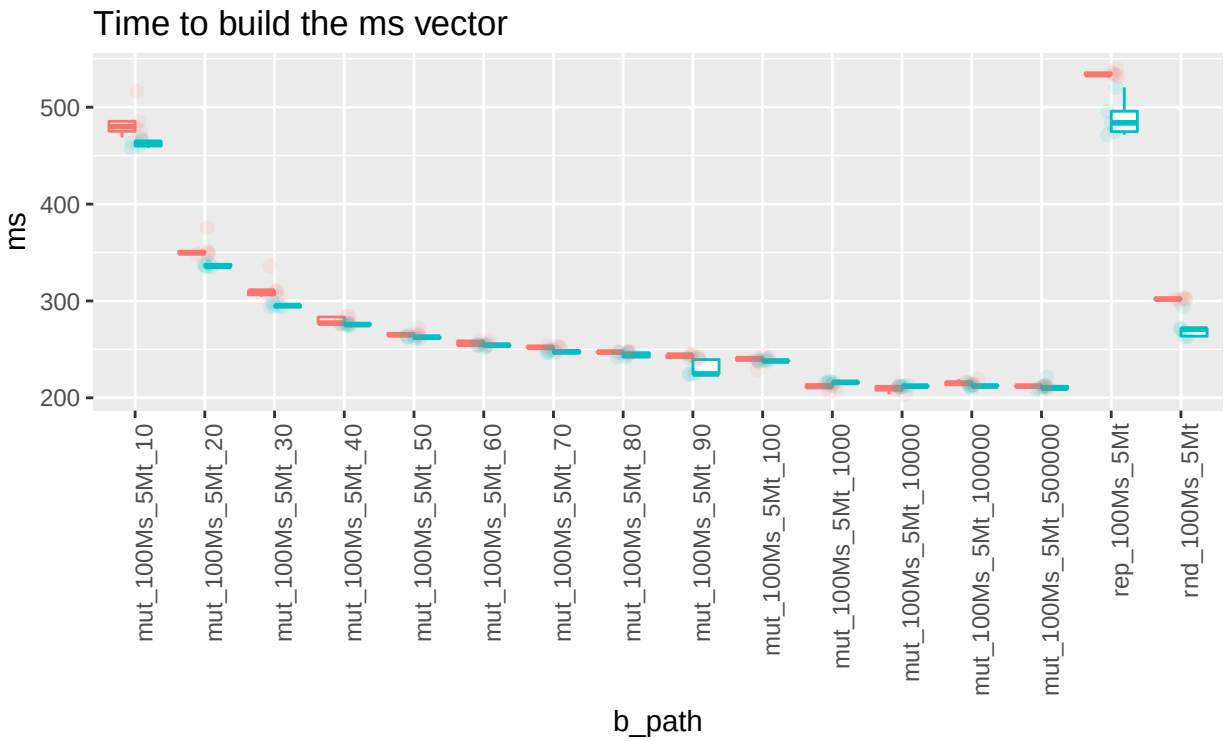
TODO: describe dataset and tests

Table 5: Sandbox performance of the two tricks

interval_width	wl_presence	double_rank_fail	double_rank_no_fail	single_rank
different_block	has_wl	694.03	692.11	777.76
different_block	no_wl	477.60	476.30	582.80
same_block	has_wl	618.90	621.30	706.35
same_block	no_wl	237.47	406.01	498.62



3.2.2 Full algorithm performance



b_path	default	rank_fail
mut_100Ms_5Mt_10	485.152	462.508
mut_100Ms_5Mt_20	354.672	336.752
mut_100Ms_5Mt_30	313.440	295.308
mut_100Ms_5Mt_40	279.716	275.728
mut_100Ms_5Mt_50	266.412	262.132
mut_100Ms_5Mt_60	256.404	253.964
mut_100Ms_5Mt_70	252.088	247.784
mut_100Ms_5Mt_80	247.248	244.396
mut_100Ms_5Mt_90	243.100	230.744
mut_100Ms_5Mt_100	238.664	238.448
mut_100Ms_5Mt_1000	211.312	215.856
mut_100Ms_5Mt_10000	209.040	211.968
mut_100Ms_5Mt_100000	215.316	212.564
mut_100Ms_5Mt_500000	211.780	212.464
rep_100Ms_5Mt	534.788	489.308
rnd_100Ms_5Mt	301.992	272.912

4 Maxrep

4.1 Maxrep construction

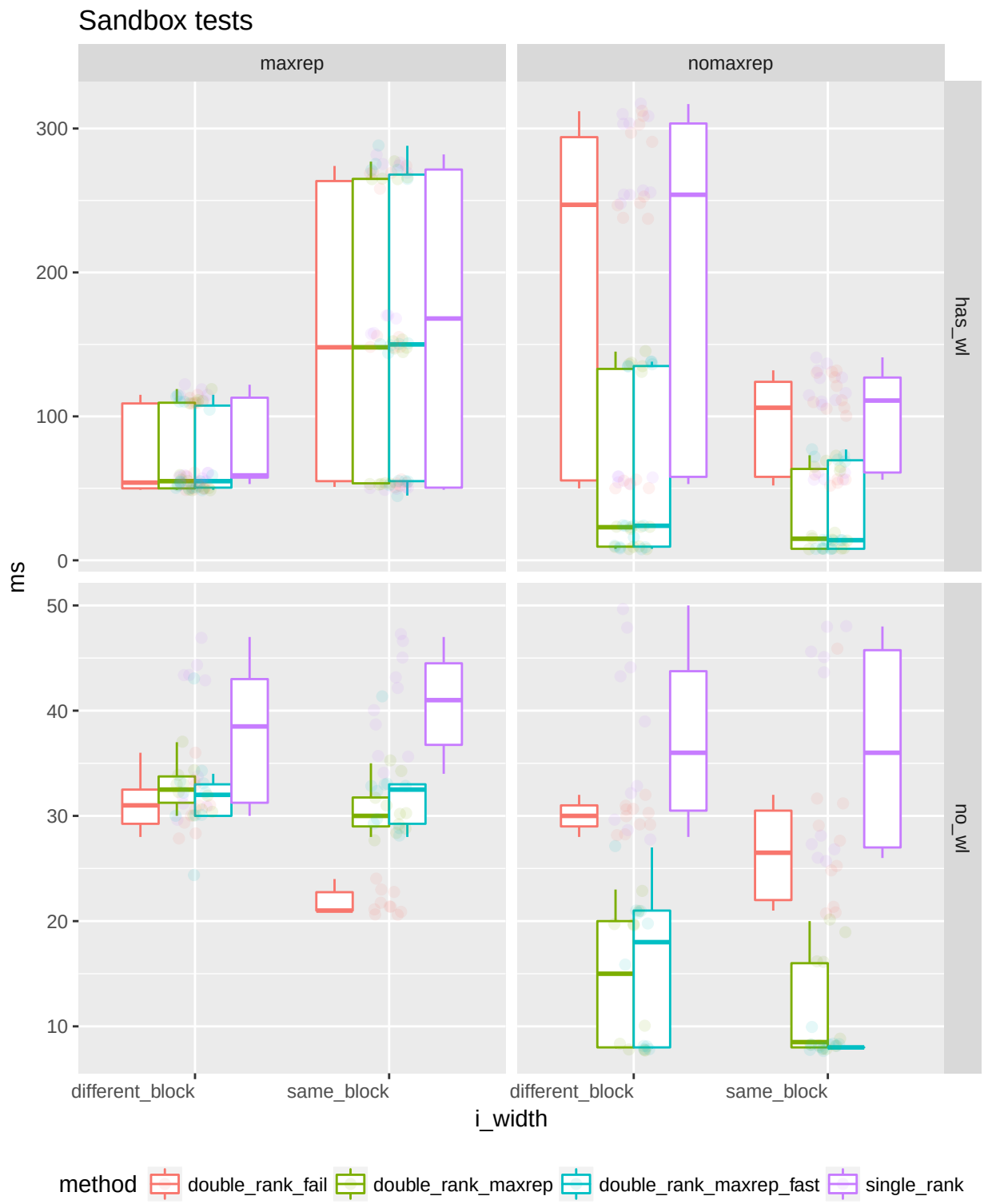
Applying the first optimization (avoid visiting subtrees of non-maximal nodes) we get 8% improvement on a (ran of a 1MB input string).

4.2 Sandbox performance

TODO: describe dataset and tests

Table 7: Sandbox performance of the two tricks

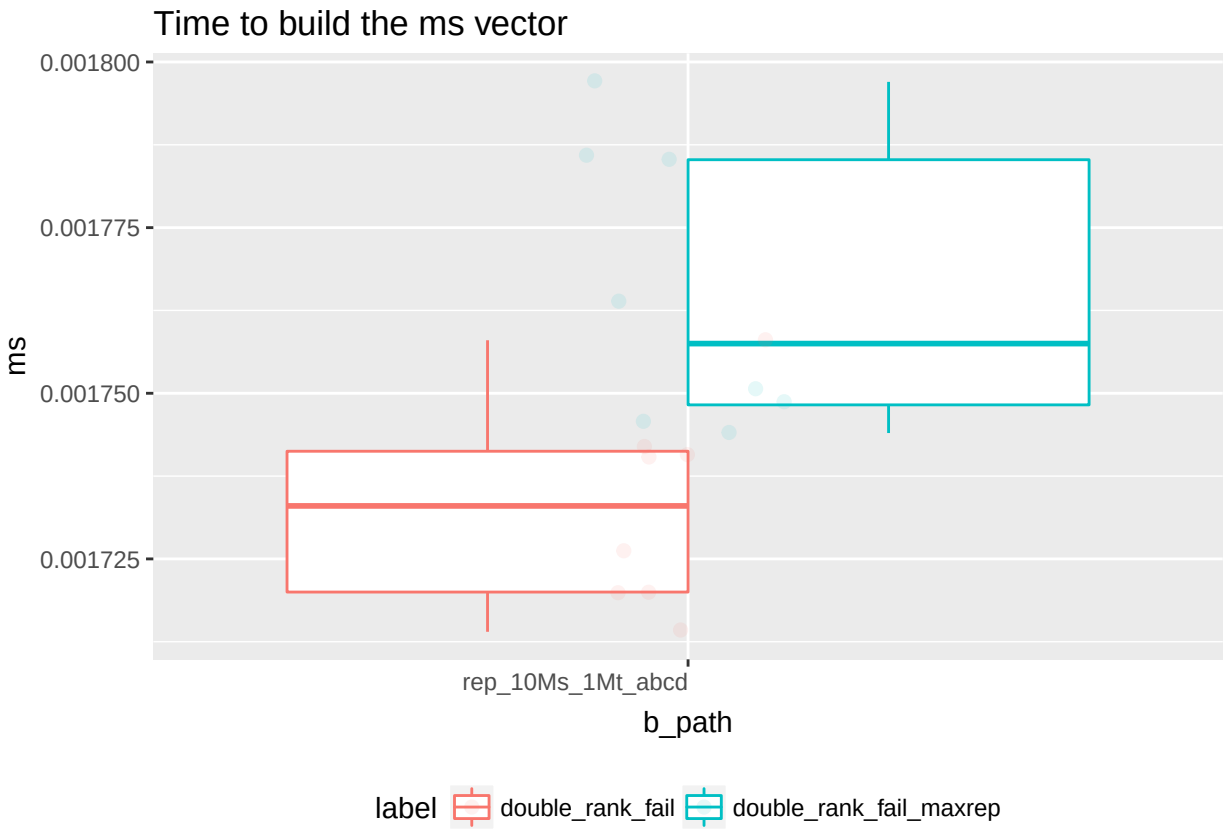
wl_presence	maximality	method	different_block	same_block
has_wl	maxrep	double_rank_fail	72.20	157.40
has_wl	maxrep	double_rank_maxrep	72.00	156.60
has_wl	maxrep	double_rank_maxrep_fast	71.93	158.40
has_wl	maxrep	single_rank	77.20	163.27
has_wl	nomaxrep	double_rank_fail	199.93	96.73
has_wl	nomaxrep	double_rank_maxrep	56.13	30.20
has_wl	nomaxrep	double_rank_maxrep_fast	55.87	31.40
has_wl	nomaxrep	single_rank	206.20	100.93
no_wl	maxrep	double_rank_fail	31.00	21.80
no_wl	maxrep	double_rank_maxrep	32.60	30.60
no_wl	maxrep	double_rank_maxrep_fast	32.10	32.10
no_wl	maxrep	single_rank	37.80	40.90
no_wl	nomaxrep	double_rank_fail	29.90	27.90
no_wl	nomaxrep	double_rank_maxrep	14.60	12.00
no_wl	nomaxrep	double_rank_maxrep_fast	15.80	8.20
no_wl	nomaxrep	single_rank	37.60	36.50


Sandbox tests
maxrep
nomaxrep
has_wl
no_wl
ms
300
200
100
0
50
40
30
20
10
different_block
same_block
different_block
same_block
i_width
method
double_rank_fail
double_rank_maxrep
double_rank_maxrep_fast
single_rank

4.3 Full Algorithm Performance

```
## Warning in data.frame(b_path = b_path, mu_nr =
## as.numeric(simplify2array(lapply(strsplit(b_path, : NAs introduced by
## coercion
```

The figure below shows 8 runs of the program with and without the use of the `maxrep` (or `B`) vector. The plot shows times (in seconds) for the construction of the `ms` bitvector. The table below that, shows the time (in seconds) to construct the `maxrep` vector. The input data is random and has |s|=10MB and |t|=1MB.



Check

```
## Source: local data frame [32 x 6]
## Groups: method, char, maximality, wl_presence [?]
##
##              method  char maximality wl_presence iwidth time_ms
##               <chr> <chr>      <chr>       <chr>  <chr>   <dbl>
## 1  double_rank_fail     a     maxrep        nowl   wide    31.8
## 2  double_rank_fail     a     maxrep          wl narrow    53.2
## 3  double_rank_fail     a   nomaxrep        nowl   wide    31.0
## 4  double_rank_fail     a   nomaxrep          wl narrow   106.4
## 5  double_rank_fail     b     maxrep          wl narrow   268.0
## 6  double_rank_fail     b     maxrep          wl   wide    50.8
## 7  double_rank_fail     b   nomaxrep        nowl narrow    22.6
## 8  double_rank_fail     b   nomaxrep          wl   wide    52.8
## 9  double_rank_fail     c     maxrep          wl narrow   151.0
## 10 double_rank_fail     c     maxrep          wl   wide    54.8
## # ... with 22 more rows
```

```
## Warning: Too few values at 555 locations: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
## 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ...
```

```
## # A tibble: 32 × 6
##     char maximality wl_presence iwidth  value  count
##    <chr>      <chr>       <chr>  <chr>  <int>  <int>
## 1      a     maxrep        nowl narrow    379    379
## 2      a     maxrep        nowl   wide   3402   3402
## 3      a     maxrep          wl narrow    727    727
## 4      a     maxrep          wl   wide   3626   3626
## 5      a  nonmaxrep        nowl narrow  20031  20031
## 6      a  nonmaxrep        nowl   wide  40290  40290
## 7      a  nonmaxrep          wl narrow  60337  60337
## 8      a  nonmaxrep          wl   wide 122991 122991
## 9      b     maxrep        nowl narrow    471    471
## 10     b     maxrep        nowl   wide   3332   3332
## # ... with 22 more rows
```

```
## # A tibble: 2 × 2
##               method total_time
##                <chr>      <dbl>
## 1   double_rank_fail   11.32167
## 2 double_rank_maxrep   11.34194
```

```
## # A tibble: 2 × 3
##                      label value_avg value_sd
##                      <chr>     <dbl>    <dbl>
## 1        double_rank_fail  1732.625 14.95648
## 2 double_rank_fail_maxrep  1765.250 21.11025
```

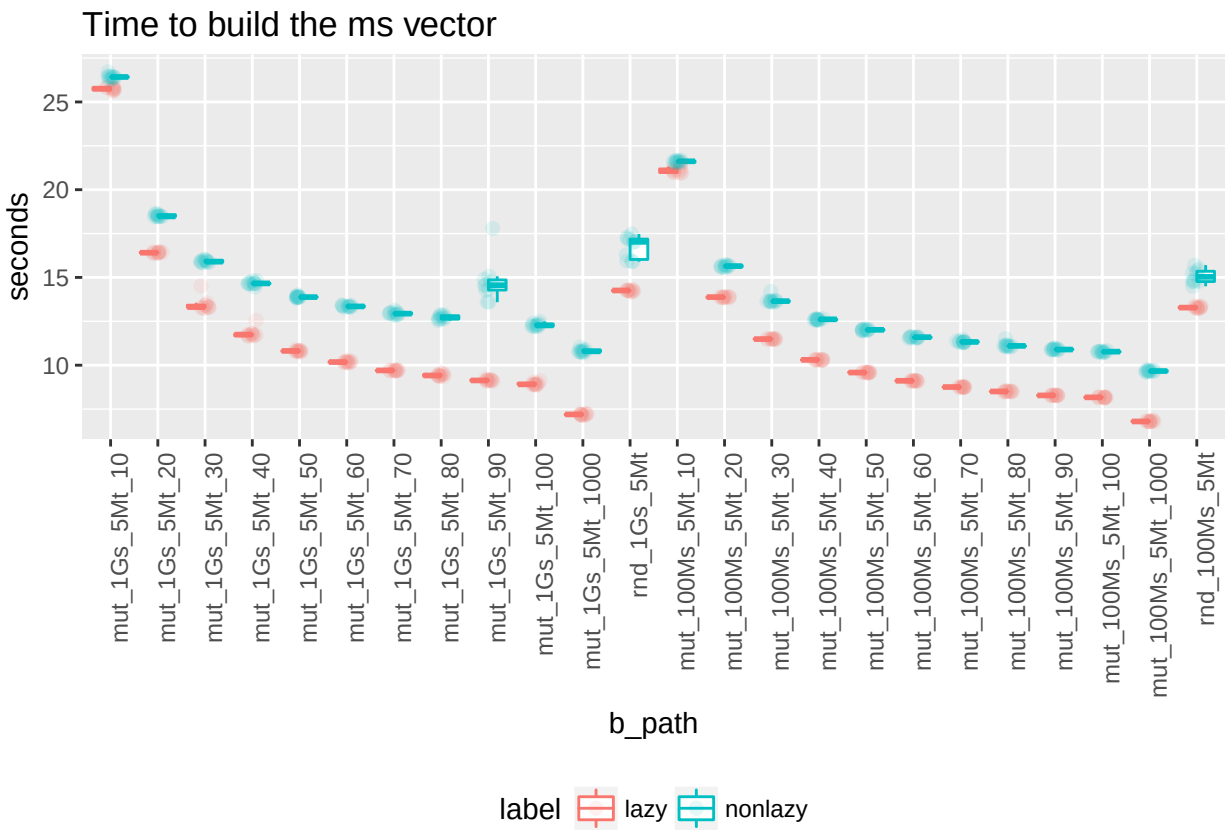
5 Lazy vs non-lazy

5.1 Code

The lazy and non-lazy versions differ in a couple of lines of code as follows

```
if(flags.lazy){
    for(; I.first <= I.second && h_star < ms_size; ){
        c = t[h_star];
        I = bstep_interval(st, I, c); //I.bstep(c);
        if(I.first <= I.second){
            v = st.lazy_wl(v, c);
            h_star++;
        }
    }
    if(h_star > h_star_prev) // // we must have called lazy_wl(). complete the node
        st.lazy_wl_followup(v);
} else { // non-lazy weiner links
    for(; I.first <= I.second && h_star < ms_size; ){
        c = t[h_star];
        I = bstep_interval(st, I, c); //I.bstep(c);
        if(I.first <= I.second){
            v = st.wl(v, c);
            h_star++;
        }
    }
}
```

5.2 Performance



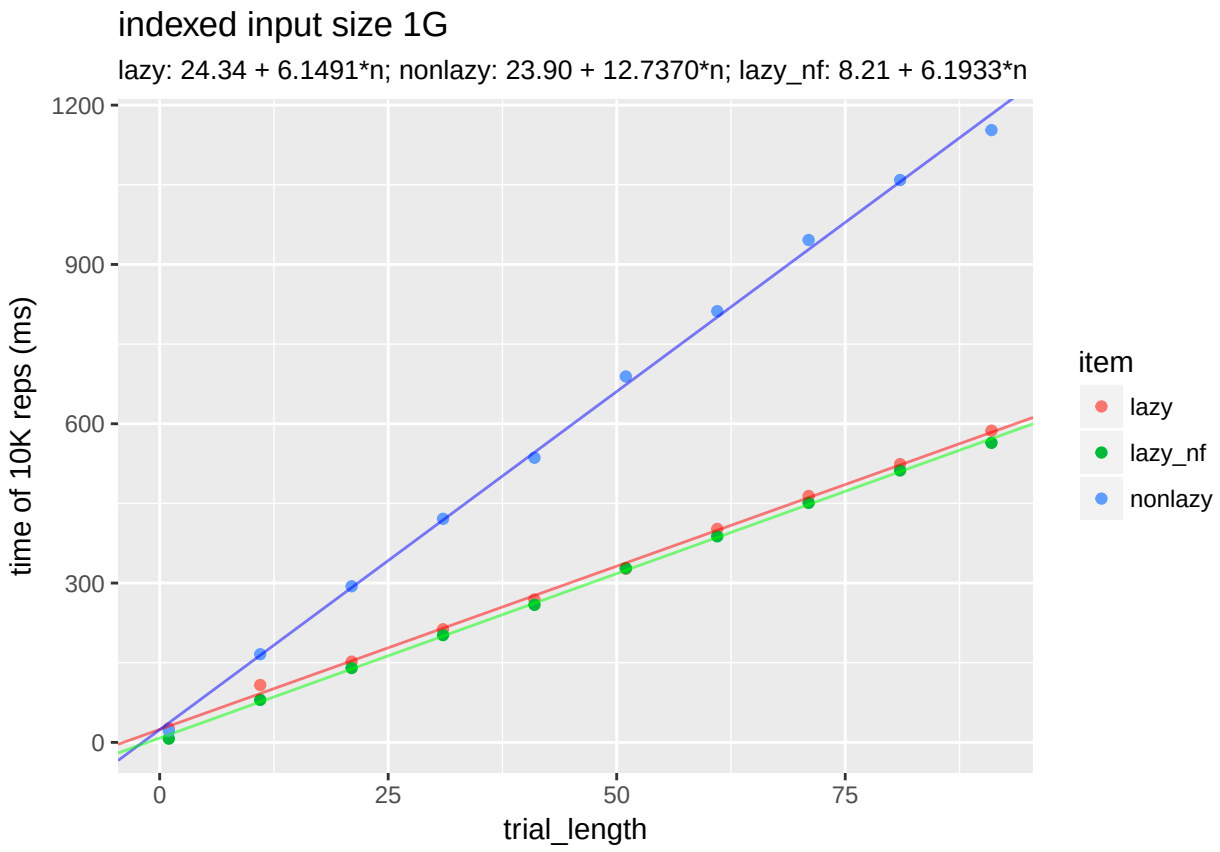
The right panel shows the time to construct the `runs` vector. This stage is the same for both versions and is shown as a control. On the left panel it can be seen that speedup correlates positively with both the size of the indexed string and the mutation period.

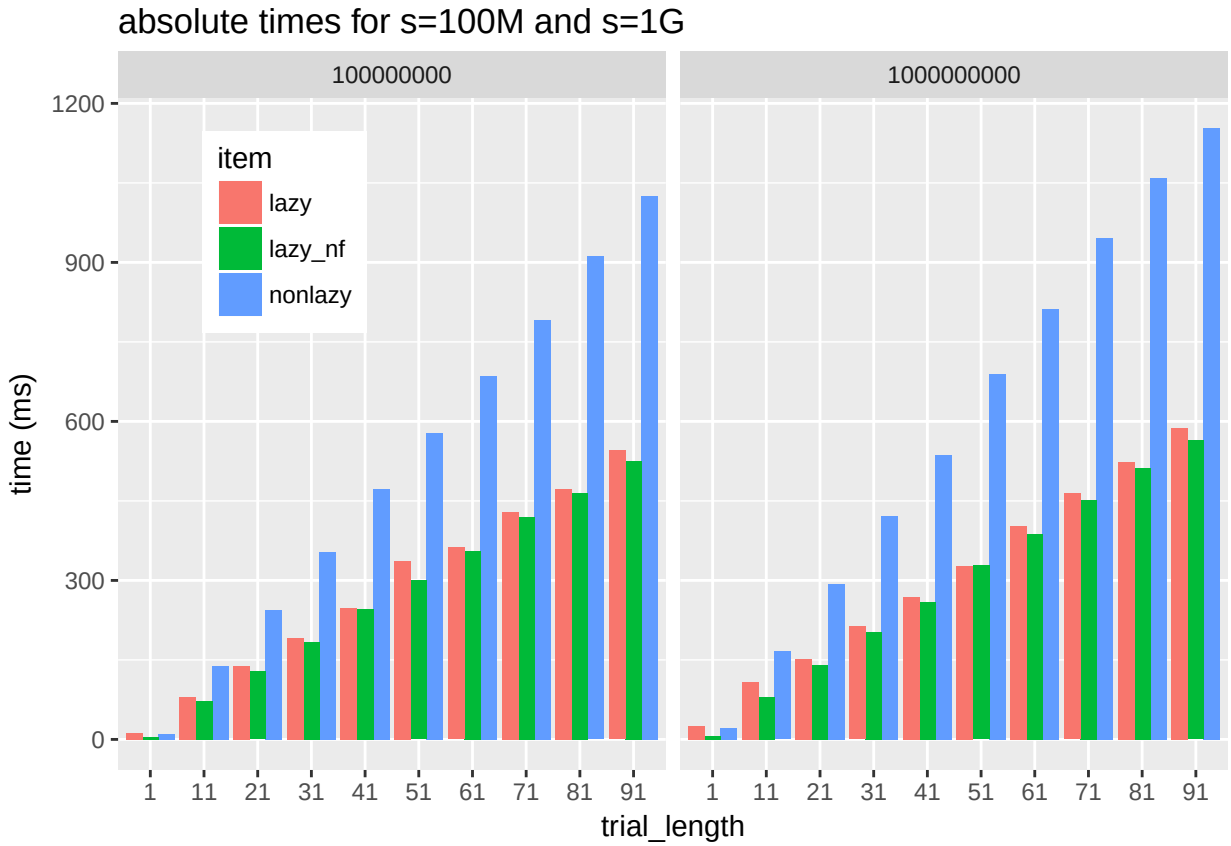
5.3 Sandbox timing

Measure the time of 10k repetitions of

- (lazy) n consecutive `lazy_wl()` calls followed by a `lazy_wl_followup()`
- (nonlazy) n consecutive `wl()` calls
- (lazy_nf) n consecutive `lazy_wl()` calls

```
// lazy
for(size_type i = 0; i < trial_length; i++)
    v = st.lazy_wl(v, s_rev[k--]);
if(h_star > h_star_prev) // // we must have called lazy_wl(). complete the node
    st.lazy_wl_followup(v);
...
// non-lazy
for(size_type i = 0; i < trial_length; i++)
    v = st.wl(v, s_rev[k--]);
...
// lazy_nf
for(size_type i = 0; i < trial_length; i++)
    v = st.lazy_wl(v, s_rev[k--]);
```





5.4 Check

In the experiments above we ran the program with the "lazy" or "non-lazy" flag and measured. The total time of each experiment can be written as $t_l = l_l + a$ and $t_n = l_n + a$ for the two versions respectively; only the ts being known. Furthermore, we have $\hat{l}_l$ and $\hat{l}_n$ estimations – computed by combining the time / wl call with the number of with the count of wl calls in each input (Section "Input Properties"). Hence we should expect

$$\delta t = t_l - t_n = l_l + a - l_n - a = l_l - l_n \approx \delta\hat{l} = \hat{l}_l - \hat{l}_n$$

b_path	t_l	t_n	l_l	l_n	delta_t	delta_l_hat
mut_100Ms_5Mt_10	21.12	21.61	8.56	6.16	-0.49	2.39
mut_100Ms_5Mt_100	8.16	10.77	3.36	4.33	-2.60	-0.97
mut_100Ms_5Mt_1000	6.80	9.67	2.84	4.15	-2.86	-1.31
mut_100Ms_5Mt_20	13.87	15.64	5.66	5.14	-1.77	0.52
mut_100Ms_5Mt_30	11.49	13.70	4.71	4.81	-2.21	-0.10
mut_100Ms_5Mt_40	10.31	12.60	4.22	4.64	-2.30	-0.41
mut_100Ms_5Mt_50	9.58	12.01	3.93	4.53	-2.43	-0.60
mut_100Ms_5Mt_60	9.11	11.58	3.74	4.47	-2.48	-0.72
mut_100Ms_5Mt_70	8.75	11.34	3.60	4.42	-2.59	-0.81
mut_100Ms_5Mt_80	8.51	11.13	3.50	4.38	-2.63	-0.88
mut_100Ms_5Mt_90	8.28	10.90	3.42	4.35	-2.62	-0.93
mut_1Gs_5Mt_10	25.75	26.43	7.57	6.65	-0.68	0.92
mut_1Gs_5Mt_100	8.94	12.29	3.49	4.90	-3.35	-1.41

b_path	t_l	t_n	l_l	l_n	delta_t	delta_l_hat
mut_1Gs_5Mt_1000	7.19	10.82	3.08	4.72	-3.63	-1.64
mut_1Gs_5Mt_20	16.42	18.52	5.30	5.68	-2.10	-0.37
mut_1Gs_5Mt_30	13.46	15.92	4.55	5.36	-2.46	-0.81
mut_1Gs_5Mt_40	11.81	14.66	4.17	5.20	-2.85	-1.02
mut_1Gs_5Mt_50	10.81	13.89	3.95	5.10	-3.08	-1.15
mut_1Gs_5Mt_60	10.19	13.36	3.80	5.03	-3.17	-1.24
mut_1Gs_5Mt_70	9.70	12.95	3.69	4.99	-3.26	-1.30
mut_1Gs_5Mt_80	9.43	12.72	3.61	4.95	-3.29	-1.35
mut_1Gs_5Mt_90	9.14	14.74	3.55	4.93	-5.60	-1.38
rnd_100Ms_5Mt	13.29	15.07	9.65	6.55	-1.78	3.10
rnd_1Gs_5Mt	14.25	16.72	8.20	6.92	-2.48	1.28

The numbers are not identical (process dependent factors might influence the running time of function calls), but they are correlated ($corr(\delta t, \delta\hat{l}) = 0.71$).

6 Double rank and fail

6.1 Code

```
// Given subtree_double_rank(v, i, j) -> (a.first, a.second) -- to simplify code

// DOUBLE RANK: int i, int j, char c
p = bit_path(c)
result_i, result_j = i, j;
node_type v = m_tree.root();
for (l = 0; l < path_len; ++l, p >>= 1) {
  a = subtree_double_rank(v, m_tree.bv_pos(v) + result_i, m_tree.bv_pos(v) + result_j);

  if(p&1){ // left child
      if(result_i > 0) result_i = a.first;
      if(result_j > 0) result_j = a.second;
  } else { // right child
      if(result_i > 0) result_i -= a.first;
      if(result_j > 0) result_j -= a.second;
  }
  v = m_tree.child(v, p&1); // goto child
}
return(result_i, result_j)

// DOUBLE RANK AND FAIL
p = bit_path(c)
result_i, result_j = i, j;
node_type v = m_tree.root();
for (l = 0; l < path_len; ++l, p >>= 1) {
  a = subtree_double_rank(v, m_tree.bv_pos(v) + result_i, m_tree.bv_pos(v) + result_j);

  if(p&1){ // left child
      if(result_i > 0) result_i = a.first;
      if(result_j > 0) result_j = a.second;
  } else { // right child
      if(result_i > 0) result_i -= a.first;
      if(result_j > 0) result_j -= a.second;
  }
  if(result_i == result_j) // Weiner Link call will fail
    return(0, 0)
  v = m_tree.child(v, p&1); // goto child
}
return(result_i, result_j)
```

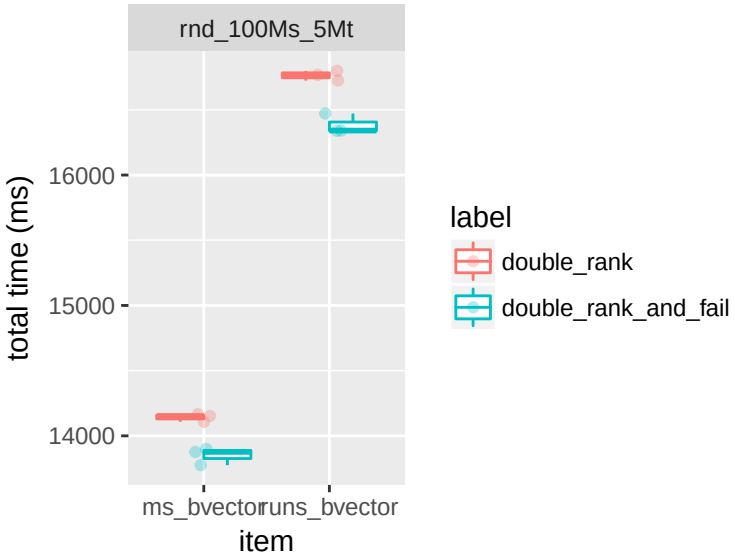
6.2 Performance

Table 9: Time (in ms) of 500K calls to `wl()` based on `single_rank()` or `double_rank()` methods on 100MB random DNA input; Mean/sd over 20 repetitions.

item	label	b_path	avg_time	sd_time
ms_bvector	double_rank	rnd_100Ms_5Mt	14142.00	30.27
ms_bvector	double_rank_and_fail	rnd_100Ms_5Mt	13850.33	66.16
runs_bvector	double_rank	rnd_100Ms_5Mt	16763.67	37.69
runs_bvector	double_rank_and_fail	rnd_100Ms_5Mt	16384.00	76.22

Table 10: Single vs. double rank. Absolute (double / single) and relative (100 * |double - single| / single) ratios of average times.

item	double_rank	double_rank_and_fail	abs_ratio	rel_ratio
ms_bvector	14142.00	13850.33	0.98	2.06
runs_bvector	16763.67	16384.00	0.98	2.26



7 Parallelization

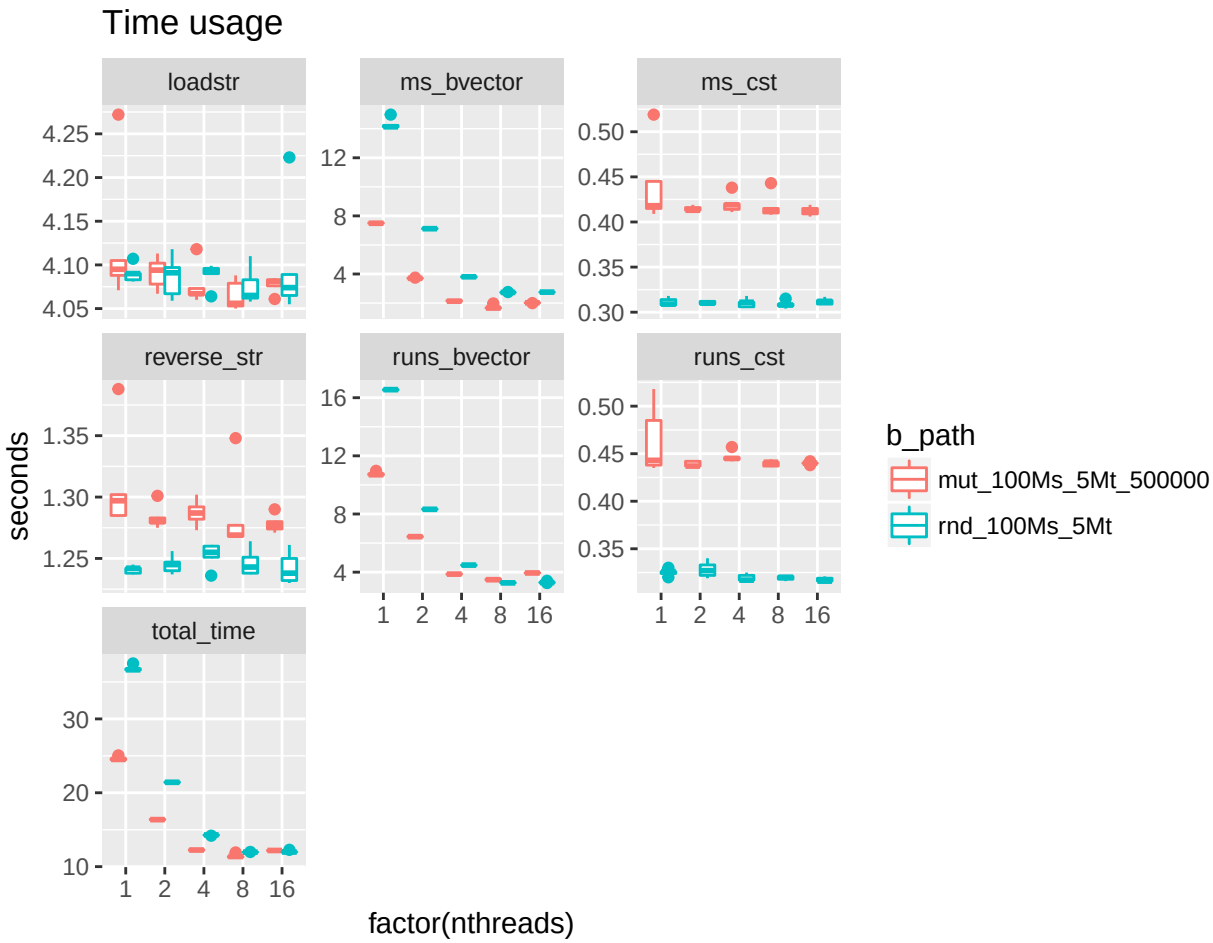
7.1 Code

See the pseudo-code in the repo (link)

7.2 Performance

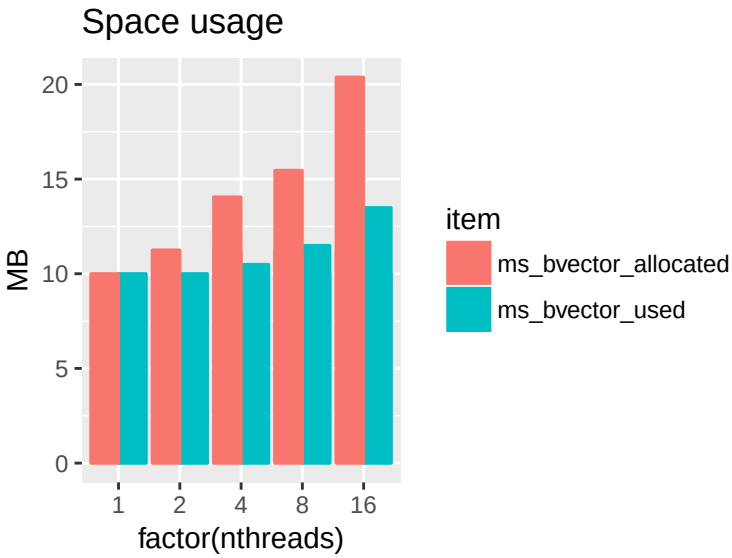
Run the MS construction program on the same input (random strings s of length 100M and t of length 5M) with varying parallelzation degree (nthreads = number of threads).

The time is reported over 5 runs for each fixed number of threads.



Space in MB for the same settings as above.

Each thread allocates its own ms vector with initial size $|t|/nthreads$ then it resizes by a factor of 1.5 each time it needs to. Resizing will always result in a vector smaller than $2|t|$ elements.


Space usage
20
15
10
5
0
MB
1
2
4
8
16
factor(nthreads)
item
ms_bvector_allocated
ms_bvector_used