New report

O. Denas

June 5, 2017

Contents

1 Current performance – not working

2 Input properties

All experiments are performed on the following inputs (the mutation period refers to the index sequence):

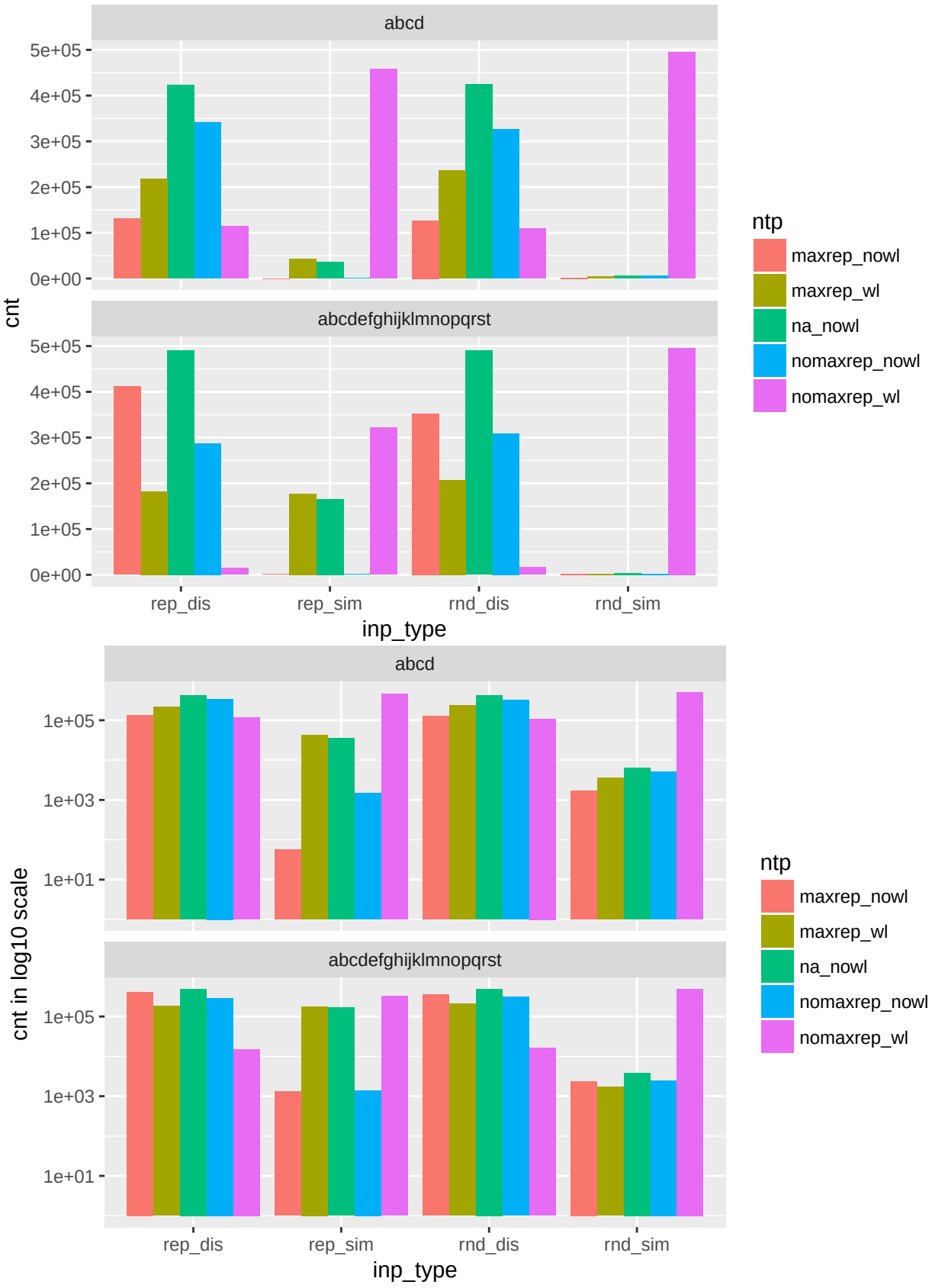
Table 1: Input datasets

s_type	t_type	slen	tlen	alp
rep	dis	100'000'000	500'000	abcdefghijklmnopqrst
rep	dis	100'000'000	500'000	abcd
rep	sim	100'000'000	500'000	abcdefghijklmnopqrst
rep	sim	100'000'000	500'000	abcd
rnd	dis	100'000'000	500'000	abcdefghijklmnopqrst
rnd	dis	100'000'000	500'000	abcd
rnd	sim	100'000'000	500'000	abcdefghijklmnopqrst
rnd	sim	100'000'000	500'000	abcd

In this section, we run the matching statistics algorithm and gather information on the way.

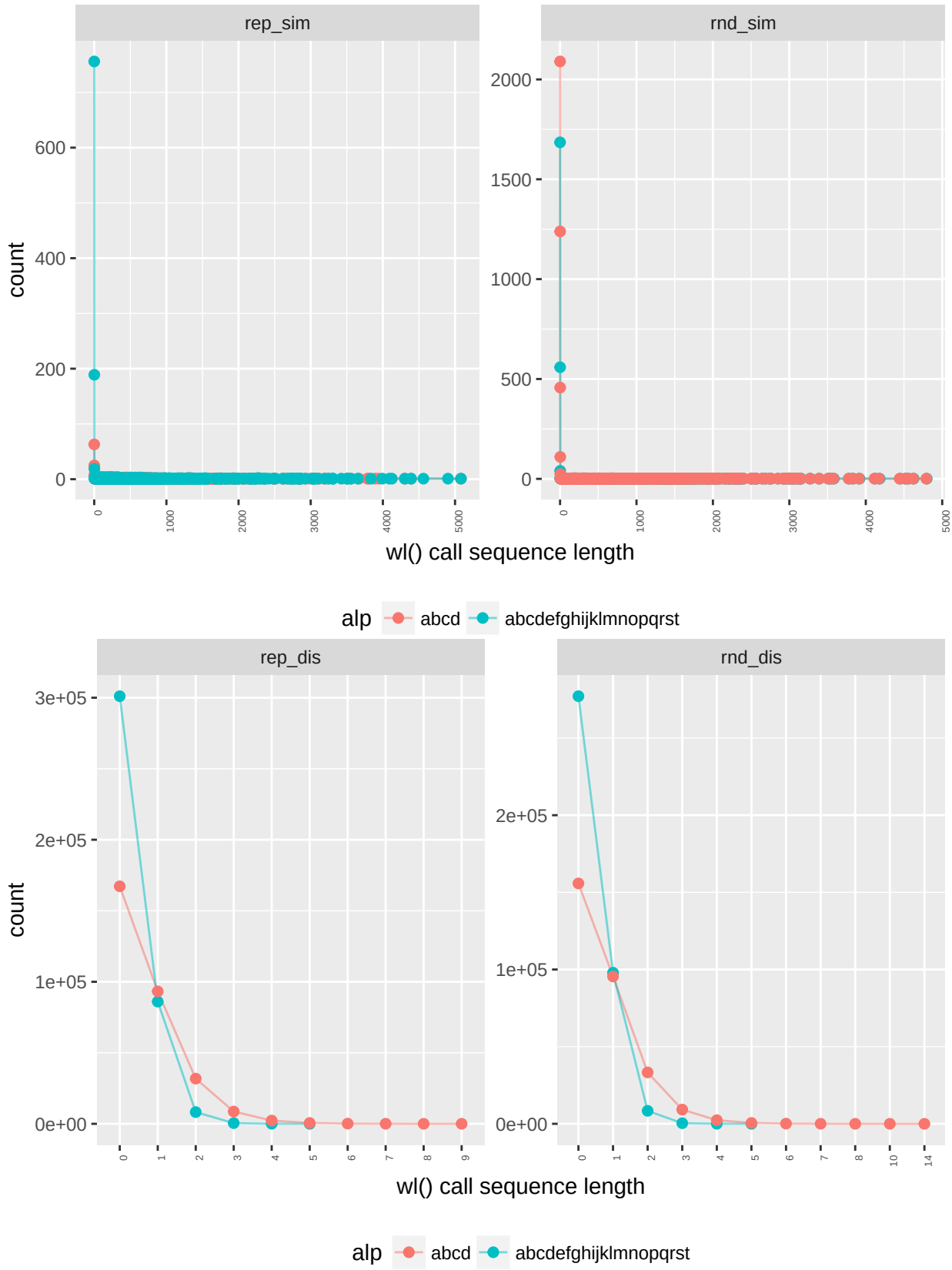
2.1 Node properties

Before making a `wl(v, c)` call register whether `v` is a maximal repeat, and whether the Weiner link exists or not.


abcd
abcdefghijklmnopqrst
cnt
5e+05
4e+05
3e+05
2e+05
1e+05
0e+00
rep_dis
rep_sim
rnd_dis
rnd_sim
inp_type
ntp
maxrep_nowl
maxrep_wl
na_nowl
nomaxrep_nowl
nomaxrep_wl
cnt in log10 scale
1e+05
1e+03
1e+01

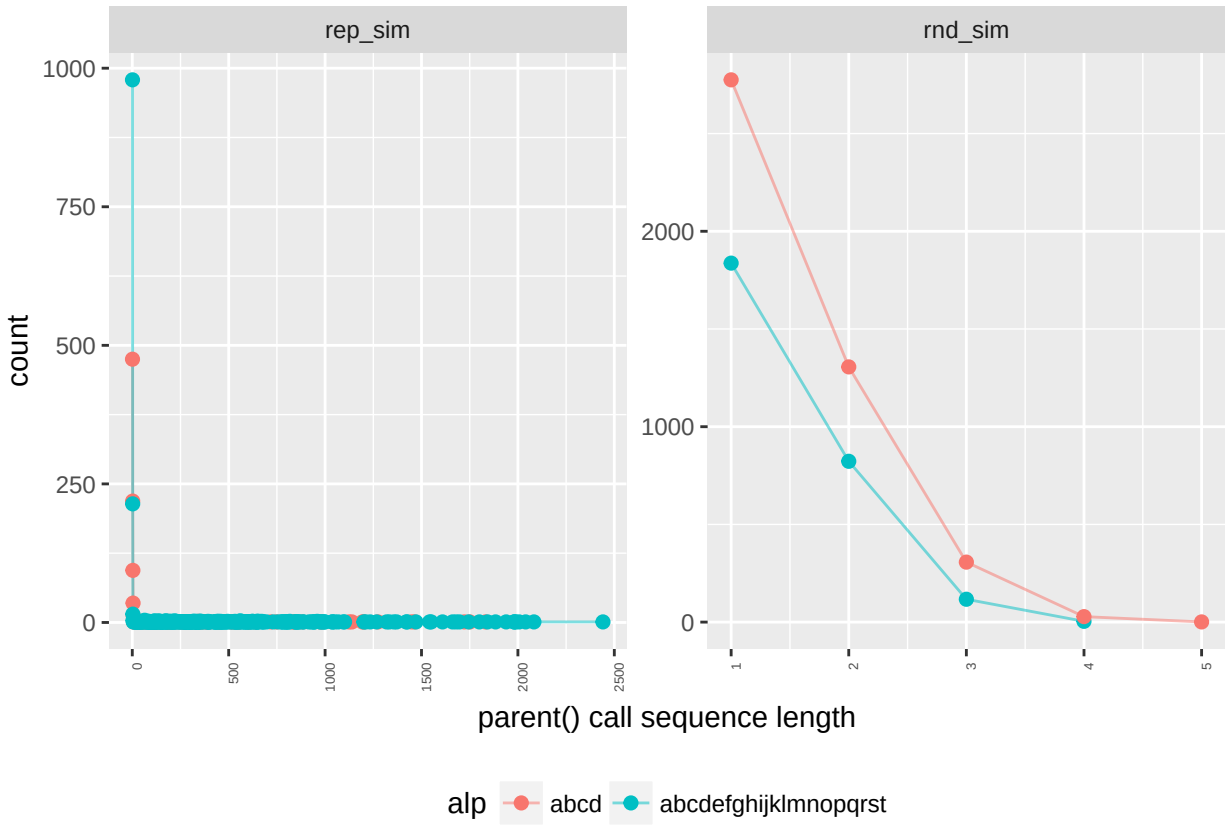
2.2 Consecutive wl() calls

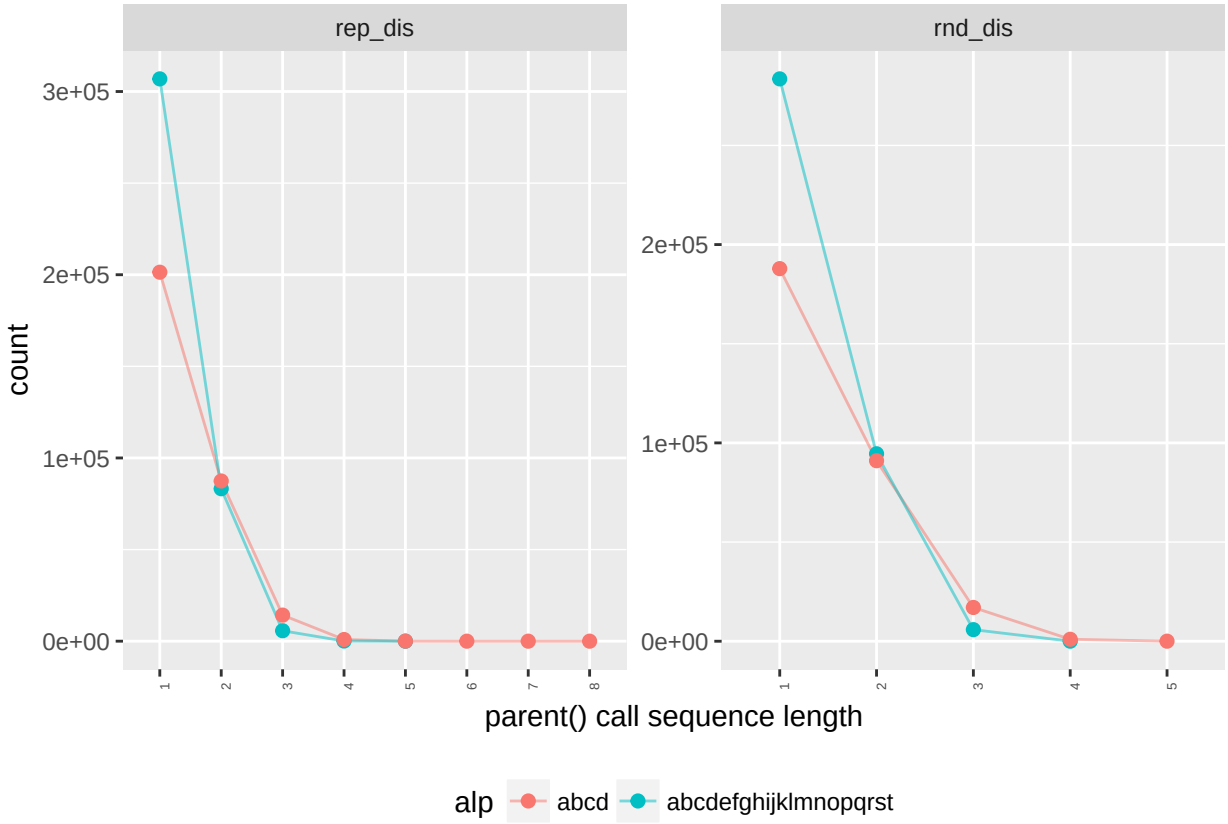
During the MS vector construction, count the number of consecutive wl() calls due to matches between reversed indexed string and the query. In other words count the k-length iterations of the while cycle.


rep_sim
rnd_sim
count
wl() call sequence length
alp
abcd
abcdefghijklmnopqrst
rep_dis
rnd_dis
count
wl() call sequence length
alp
abcd
abcdefghijklmnopqrst

2.3 Consecutive parent() calls

During the MS vector construction, count the number of consecutive `parent()` calls after a failed `wl()` call.


rep_sim
rnd_sim
count
1000
750
500
250
0
2000
1000
0
0
500
1000
1500
2000
2500
1
2
3
4
5
parent() call sequence length
alp
abcd
abcdefghijklmnopqrst

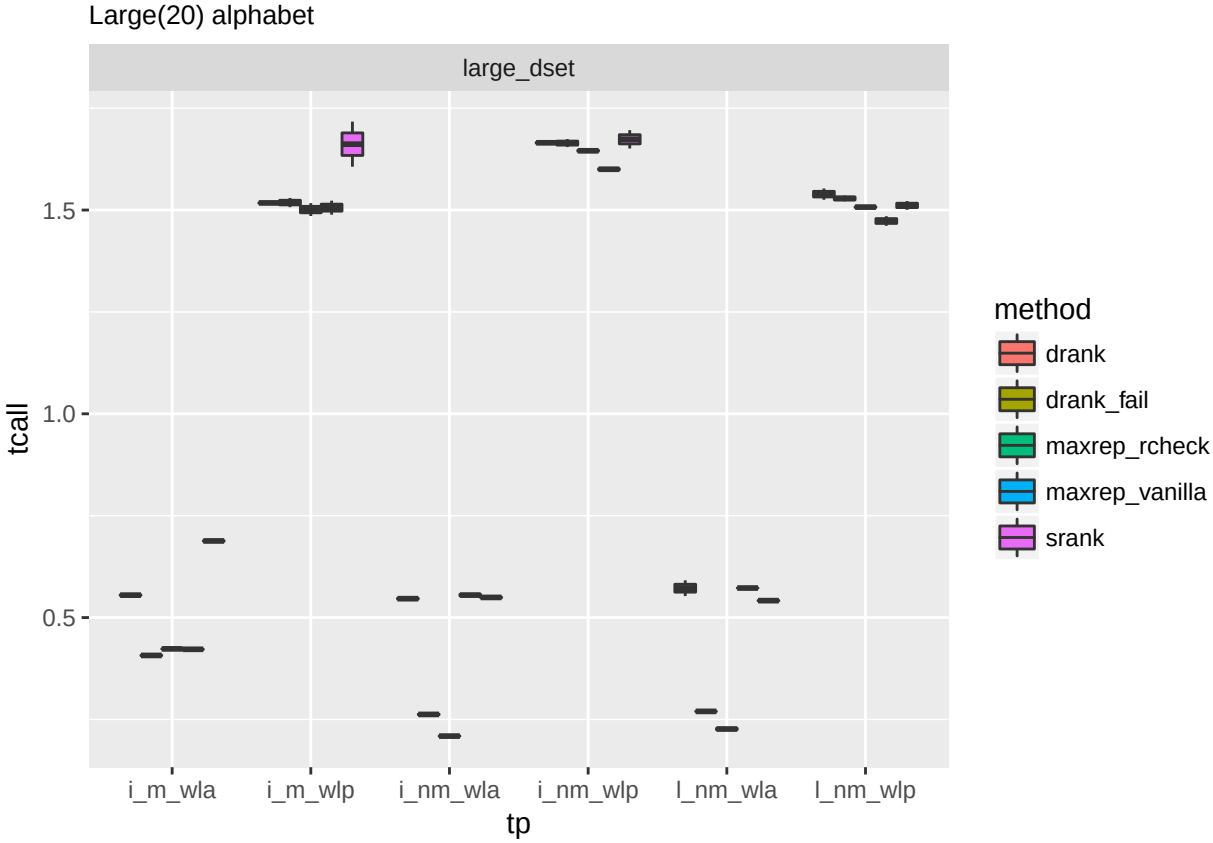

rep_dis
rnd_dis
count
3e+05
2e+05
1e+05
0e+00
2e+05
1e+05
0e+00
1
2
3
4
5
6
7
8
parent() call sequence length
alp
abcd
abcdefghijklmnopqrst

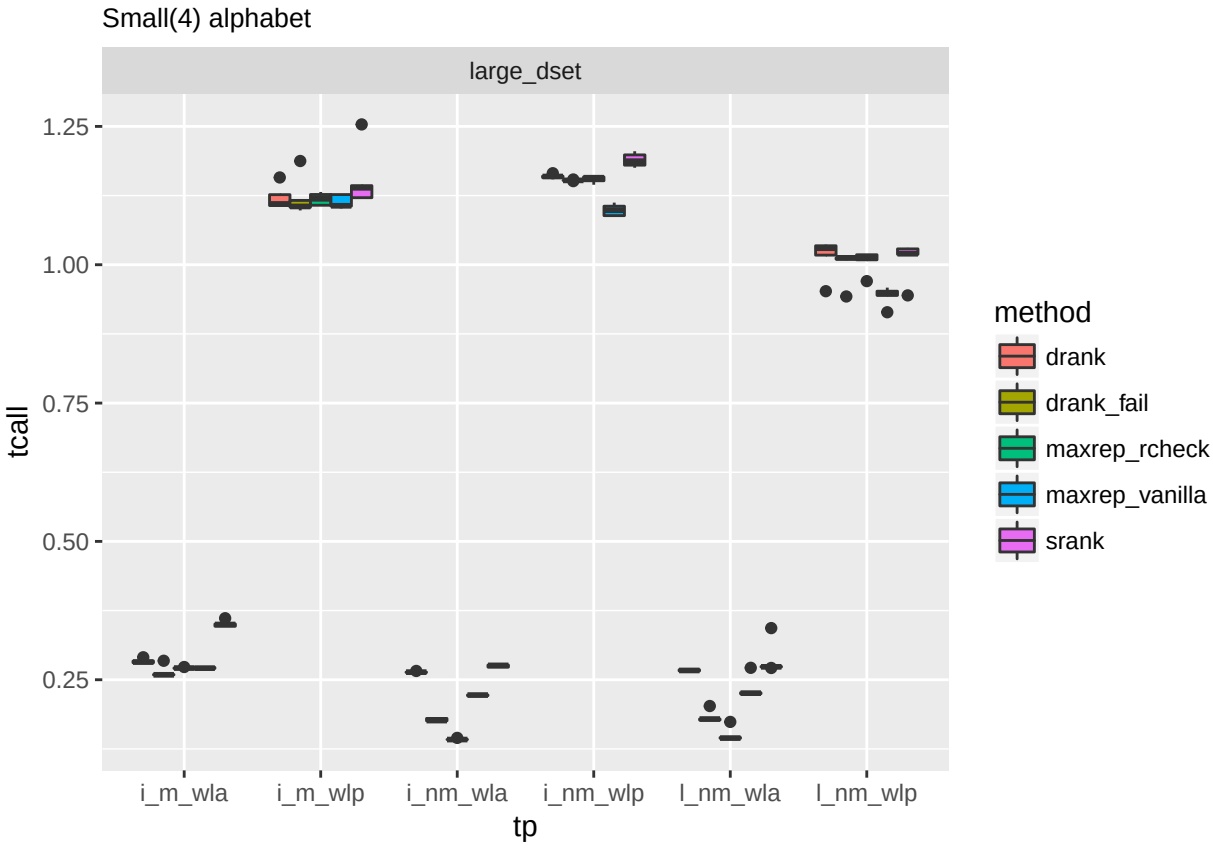
3 wl() optimizations

3.1 sandbox tests

The 4 plots correspond to combinations of index size (one 100Mb and one 1Mb) and alphabet size (one 4 and one 20). Each plot shows 6 measurements based on

- node type: leaf (l) or internal (i)
- maximality: maximal (m) or non-maximal (nm)
- WL presence: present (wlp) or absent (wla)





3.2 full tests

I run the program several times with and without the optimization. The pointrange plots report (median, with quartile ranges) the relative difference of each optimized time from the average non-optimized time in the construction time of the MS vector which is

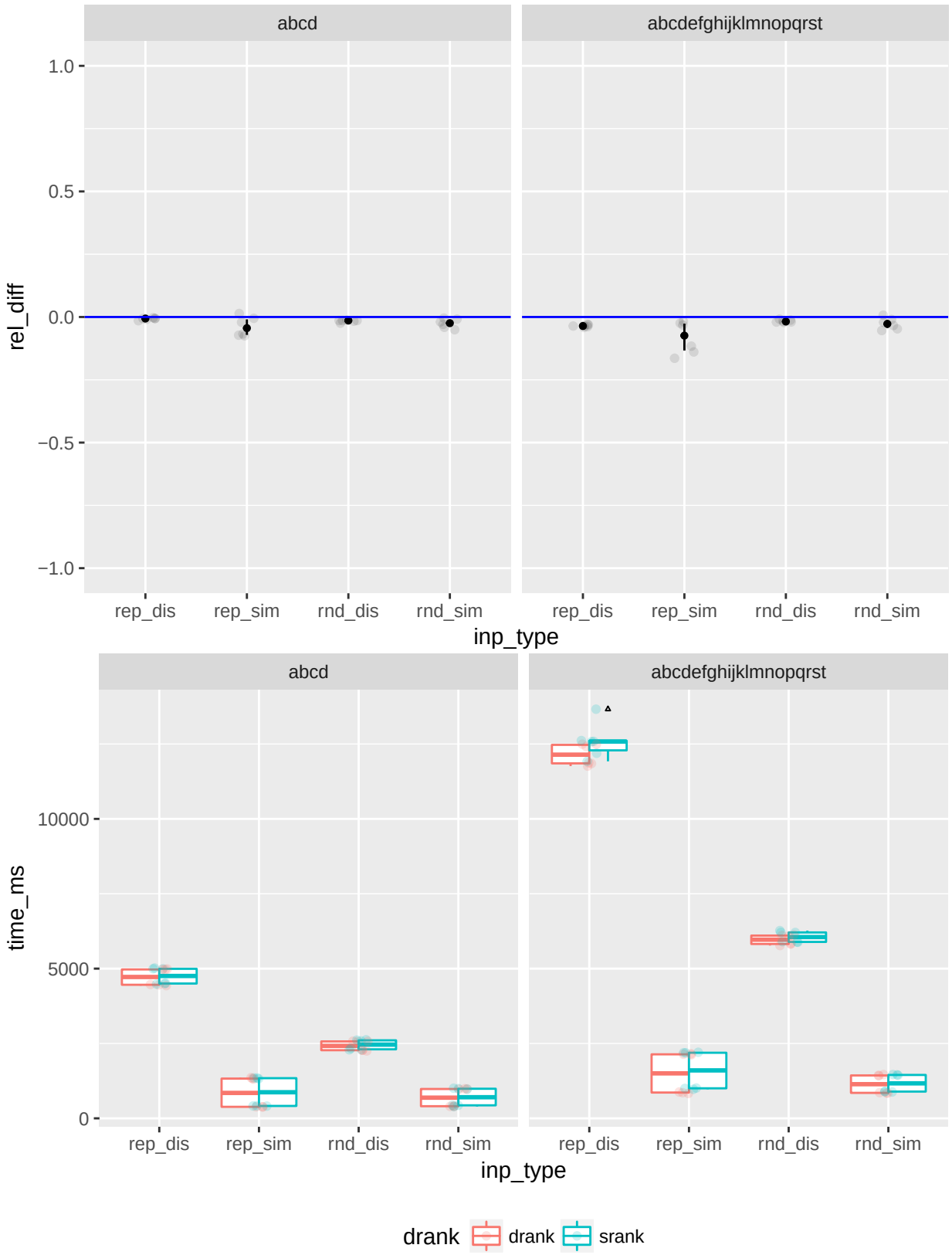
$$d^{(i)} = \frac{t^{(i)}_{\texttt{opt}} - \bar{t}_{\texttt{non\_opt}}}{\max\{t^{(i)}_{\texttt{opt}}, \bar{t}_{\texttt{non\_opt}}\}}$$

with $\bar{t}_{\texttt{non\_opt}} = 1/n \sum t^{(i)}_{\texttt{non\_opt}}$. Hence negative values indicate a speedup by the optimization over the average non-optimized time: -0.5 is a 2x speedup etc, in general the speed up is $-1/d^{(i)}$.

The boxplots report the raw times for the MS construction.

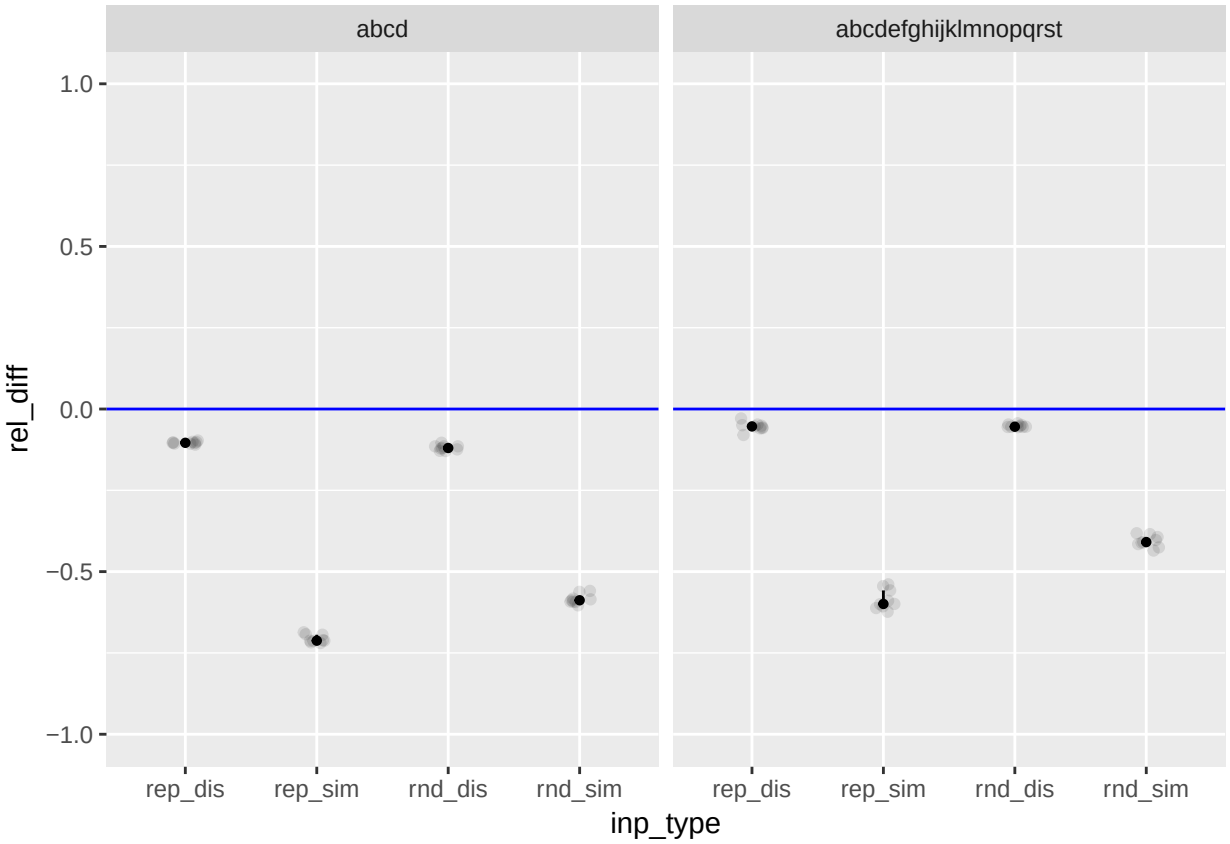
3.2.1 drank

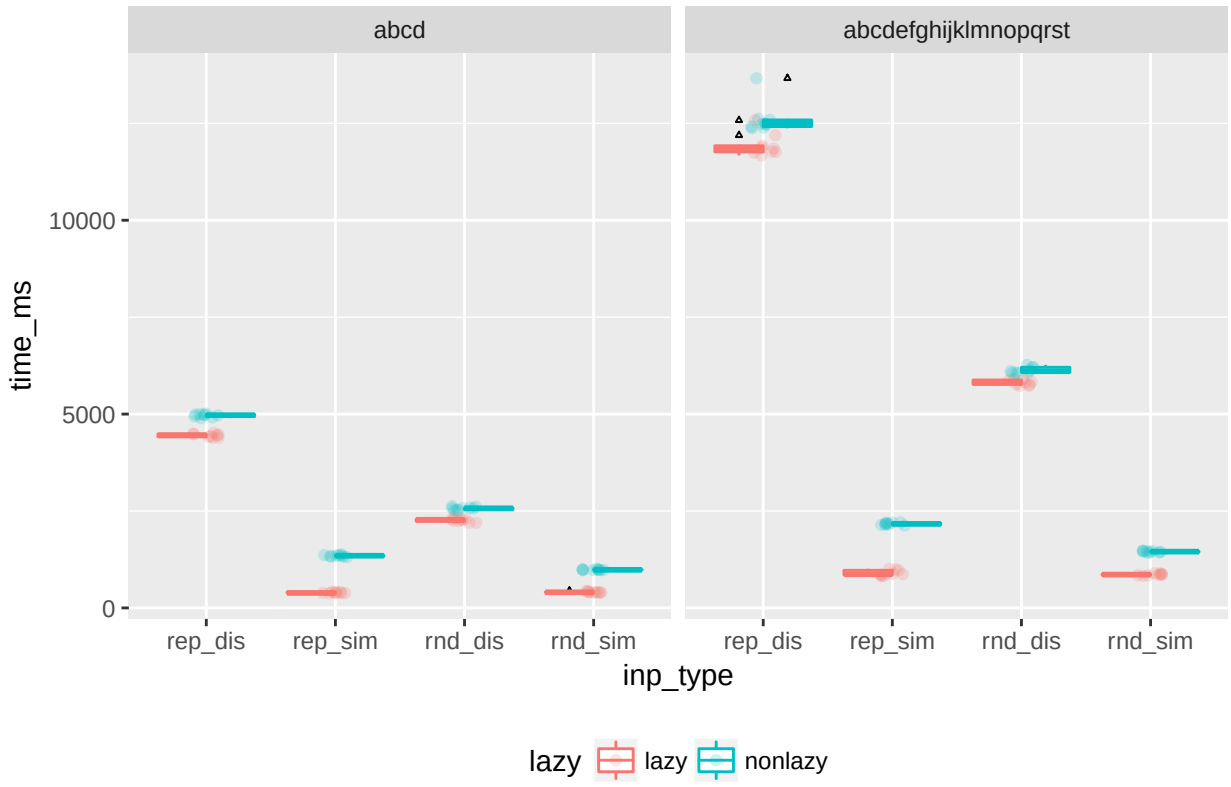
```
## # A tibble: 2 x 2
##   drank     a
##   <chr> <int>
## 1 drank    48
## 2 srank    48
```


abcd
abcdefghijklmnopqrst
rel_diff
1.0
0.5
0.0
−0.5
−1.0
rep_dis
rep_sim
rnd_dis
rnd_sim
inp_type
time_ms
10000
5000
0
drank
drank
srank

3.2.2 Lazy

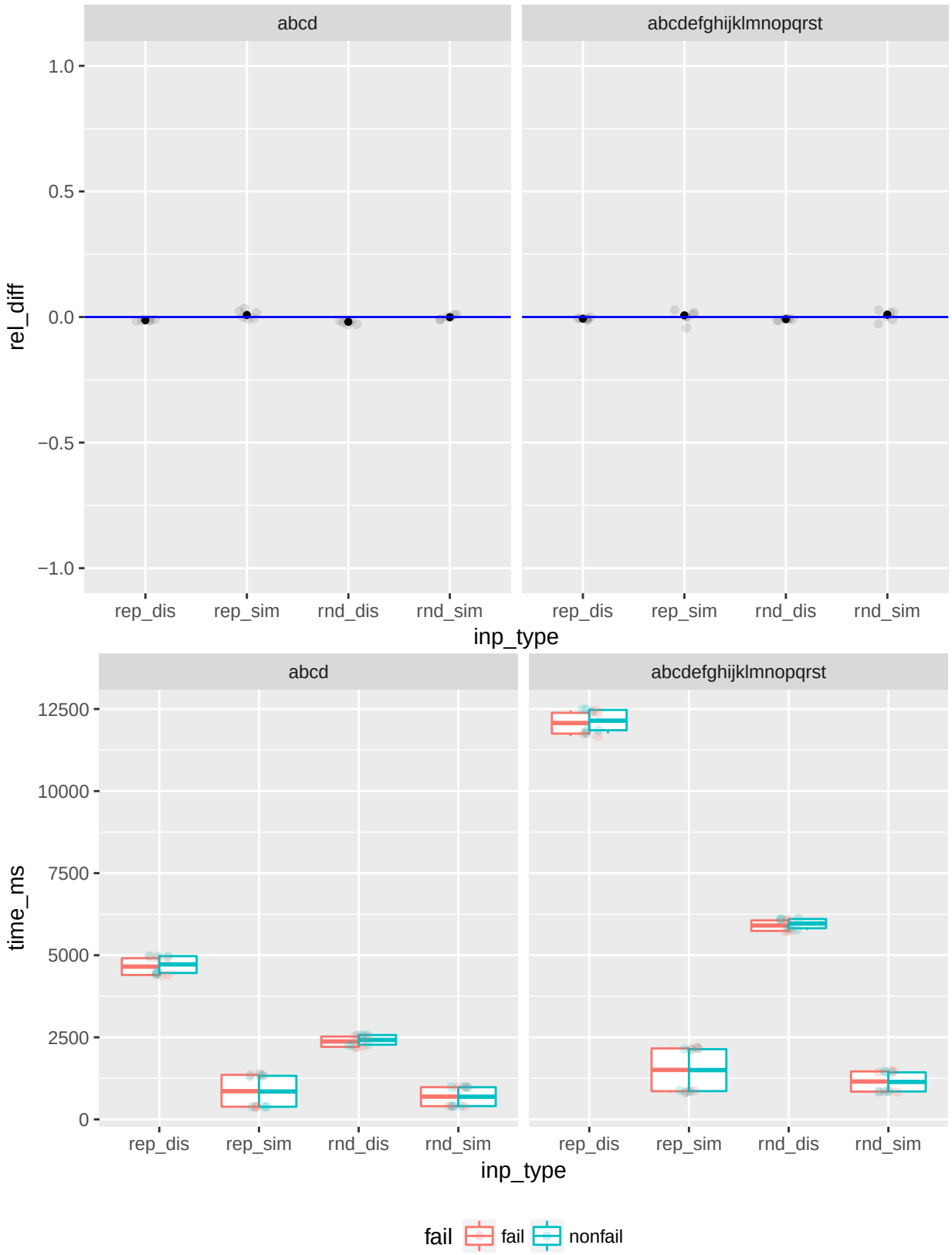
```
## # A tibble: 2 x 2
##      lazy     a
##     <chr> <int>
## 1    lazy    72
## 2 nonlazy    72
```





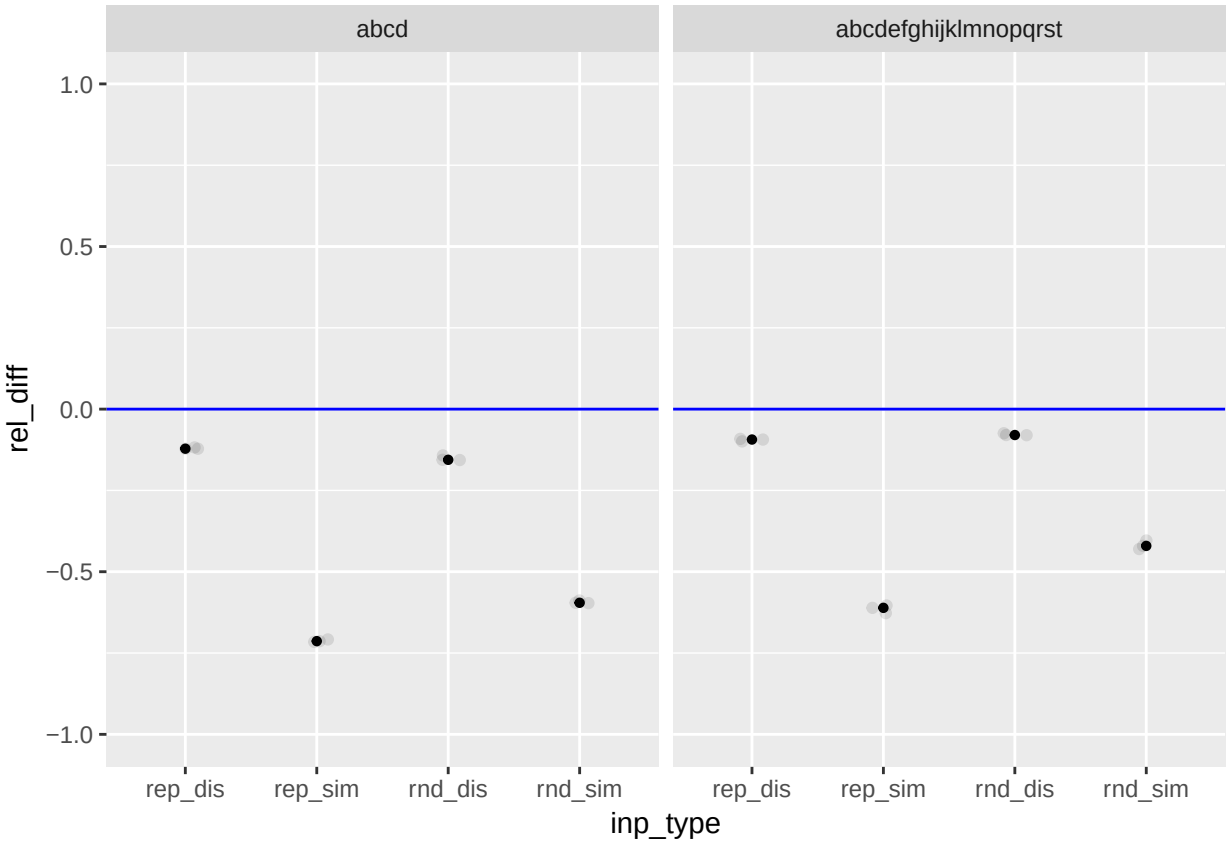
3.2.3 Fail

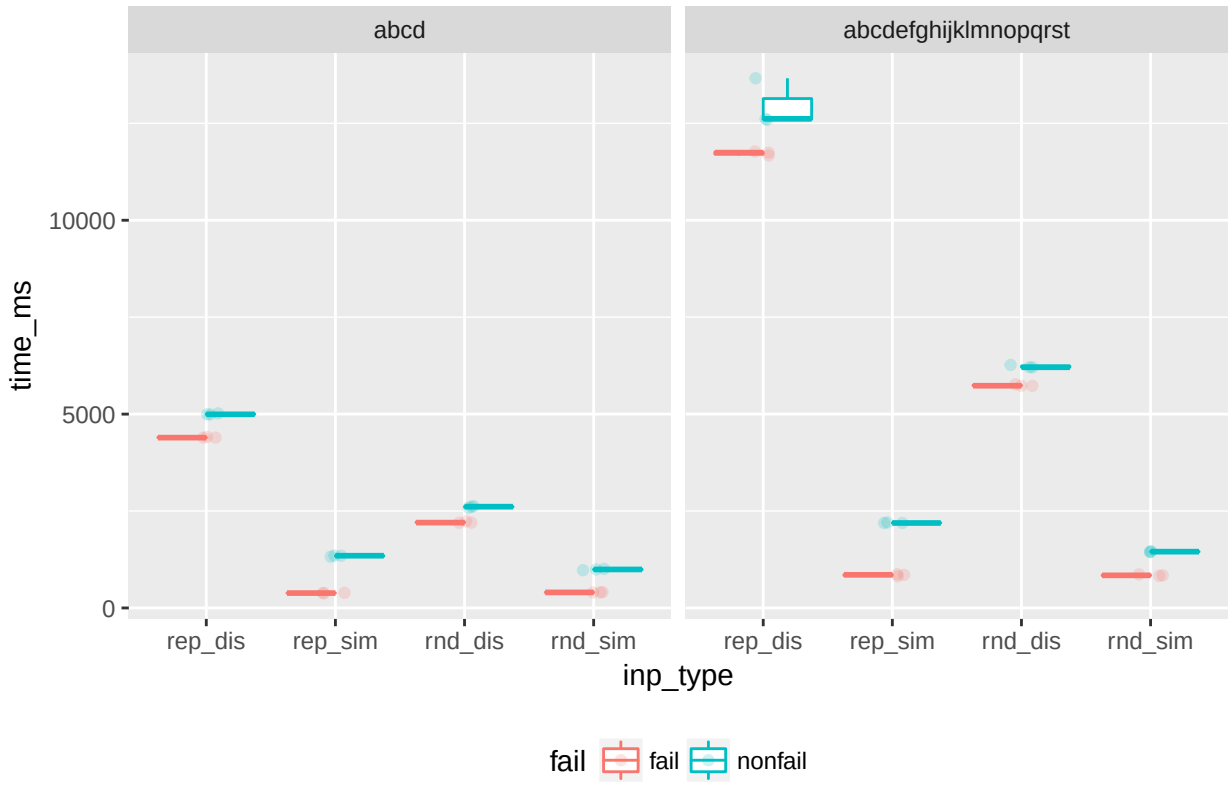
```
## # A tibble: 2 x 2
##      fail     a
##     <chr> <int>
## 1    fail    48
## 2 nonfail    48
```


abcd
abcdefghijklmnopqrst
rel_diff
1.0
0.5
0.0
−0.5
−1.0
rep_dis
rep_sim
rnd_dis
rnd_sim
inp_type
time_ms
12500
10000
7500
5000
2500
0
fail
fail
nonfail

3.2.4 all flags (lazy, fail, double_rank) vs. single rank

```
## # A tibble: 2 x 2
##   drank     a
##   <chr> <int>
## 1 drank    24
## 2 srank    24
```

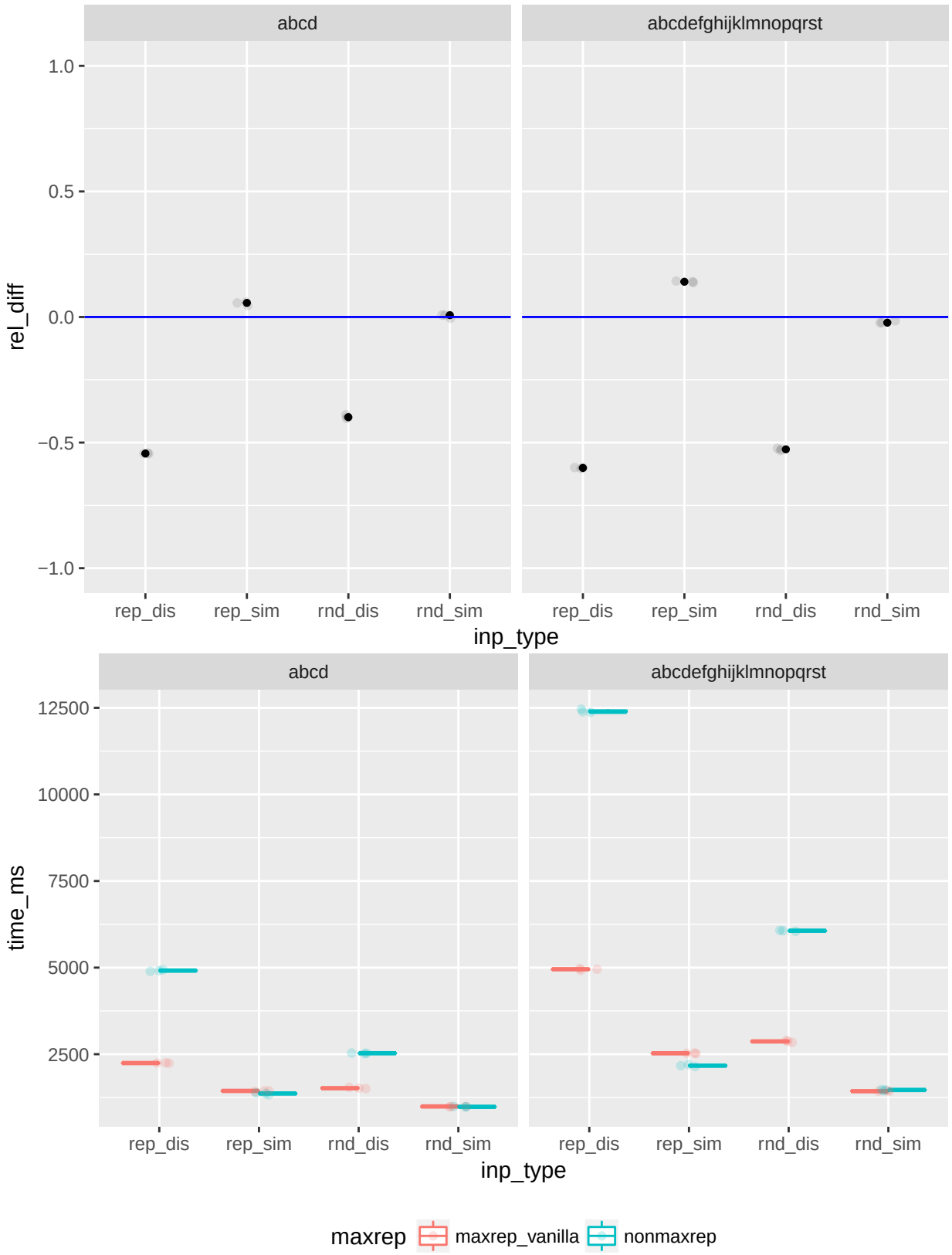




3.2.5 maxrep

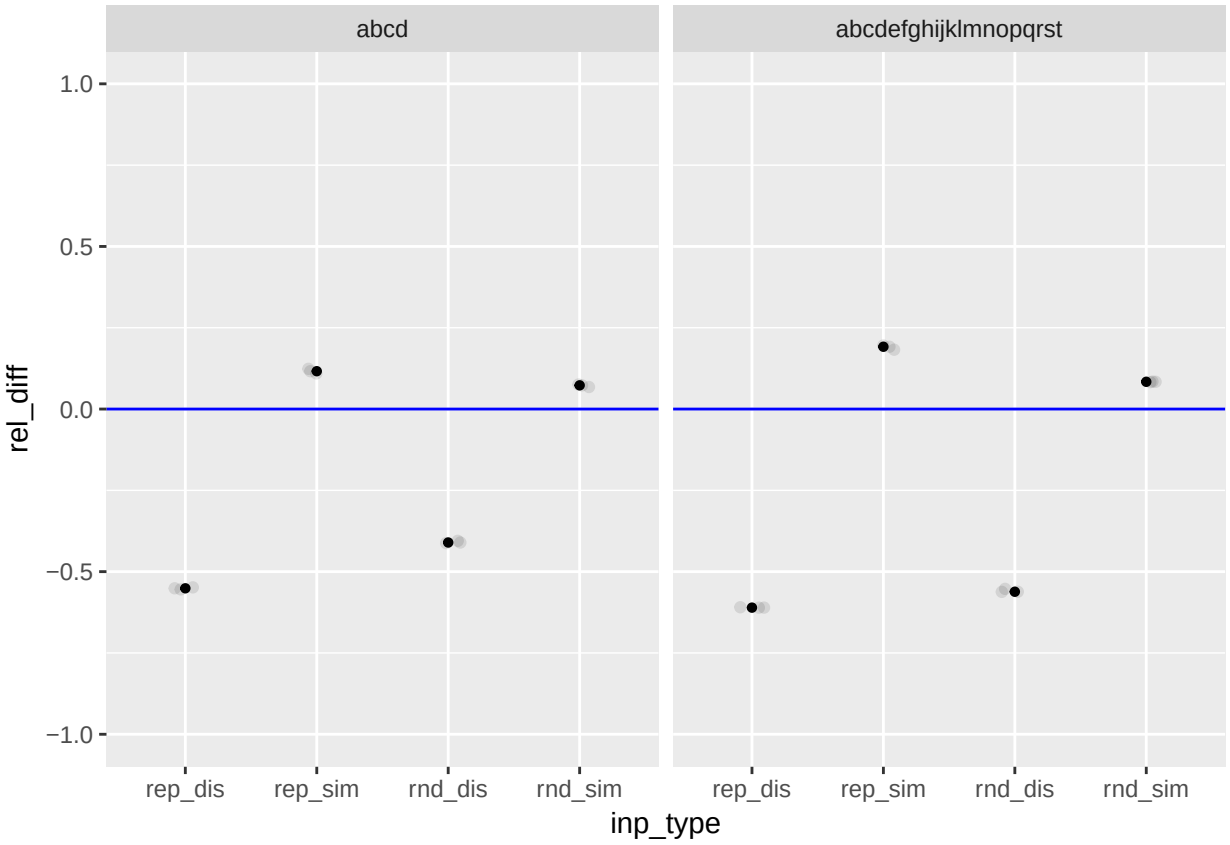
3.2.5.1 maxrep_vanilla vs. non-maxrep

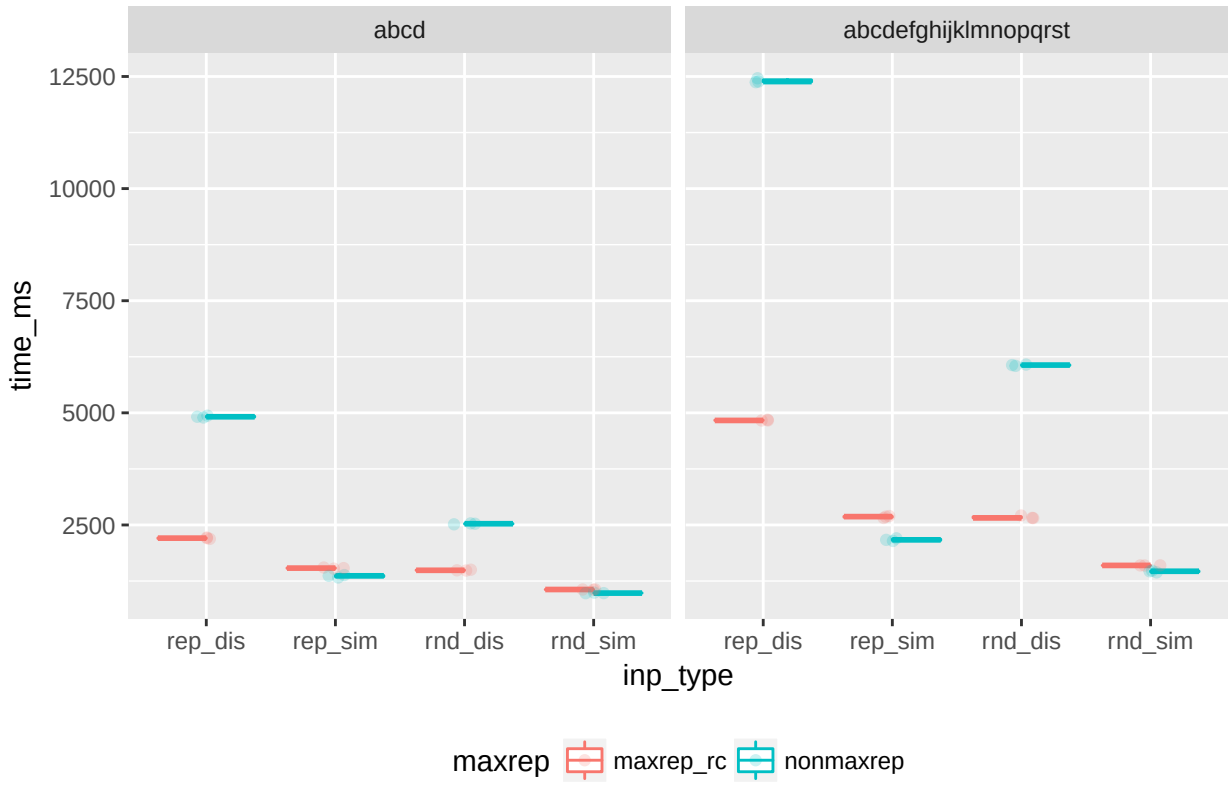
```
## # A tibble: 2 x 2
##           maxrep     a
##            <chr> <int>
## 1 maxrep_vanilla    24
## 2      nonmaxrep    24
```



3.2.5.2 maxrep+rank_and_check vs. non-maxrep

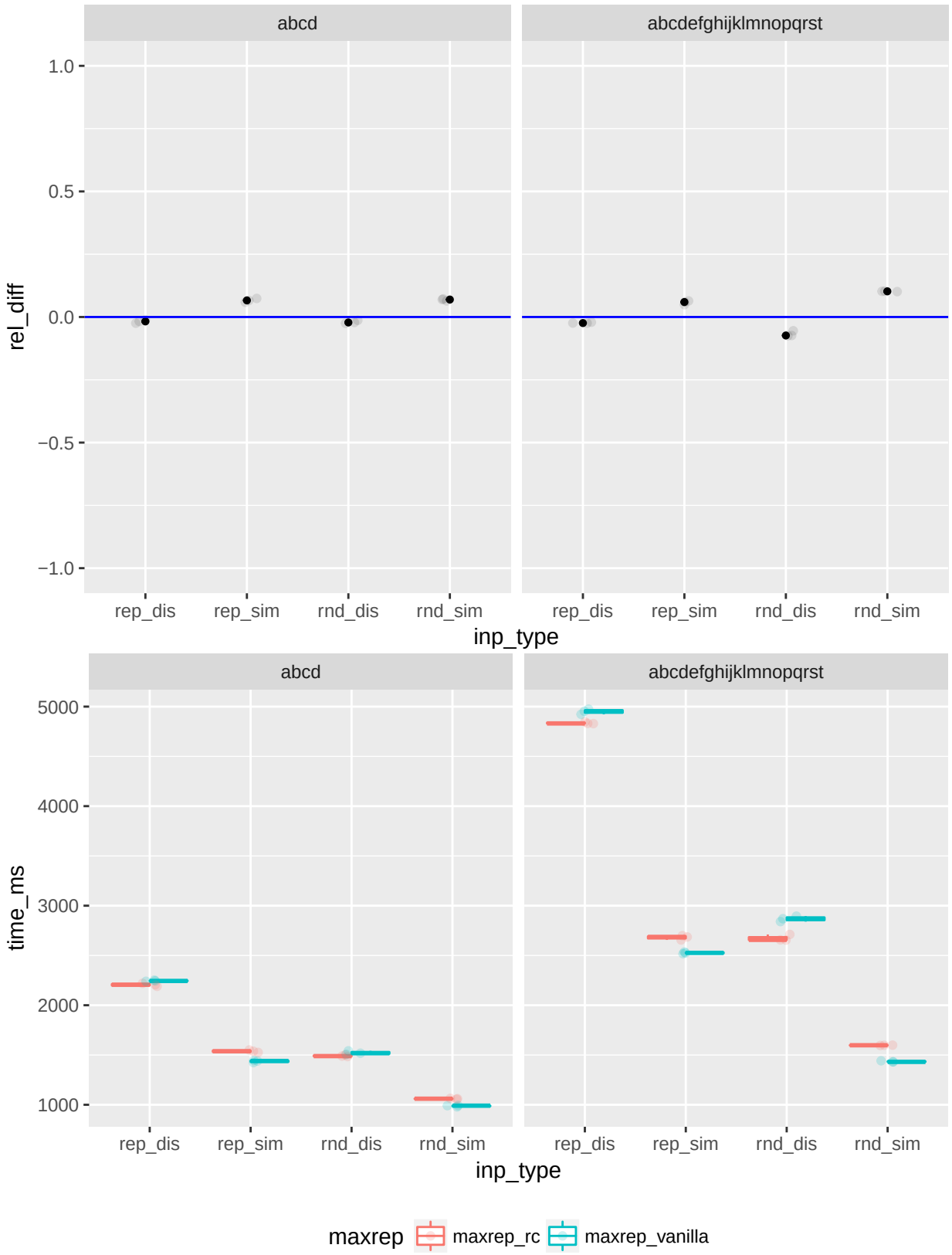
```
## # A tibble: 2 x 2
##      maxrep     a
##       <chr> <int>
## 1 maxrep_rc    24
## 2 nonmaxrep    24
```





3.2.5.3 maxrep+rank_and_check vs. maxrep_vanilla

```
## # A tibble: 2 x 2
##           maxrep     a
##            <chr> <int>
## 1      maxrep_rc    24
## 2 maxrep_vanilla    24
```

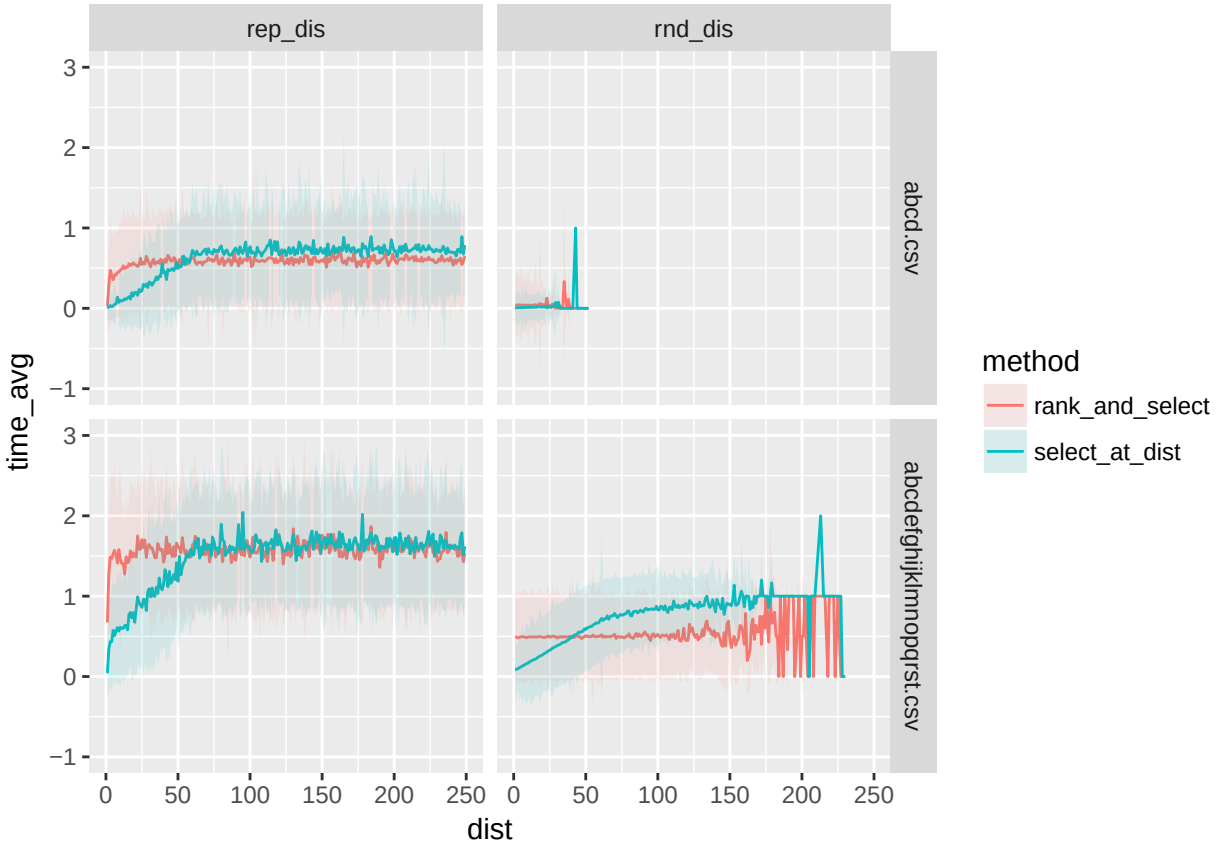

abcd
abcdefghijklmnopqrst
rel_diff
1.0
0.5
0.0
−0.5
−1.0
rep_dis
rep_sim
rnd_dis
rnd_sim
inp_type
time_ms
5000
4000
3000
2000
1000
maxrep
maxrep_rc
maxrep_vanilla

4 parent optimizations

4.1 sandbox tests

4.1.1 select at dist

Traverse the tree and time the call `select_at_dist` (labeled f) and the call `select(rank())` (labeled s) on each node.



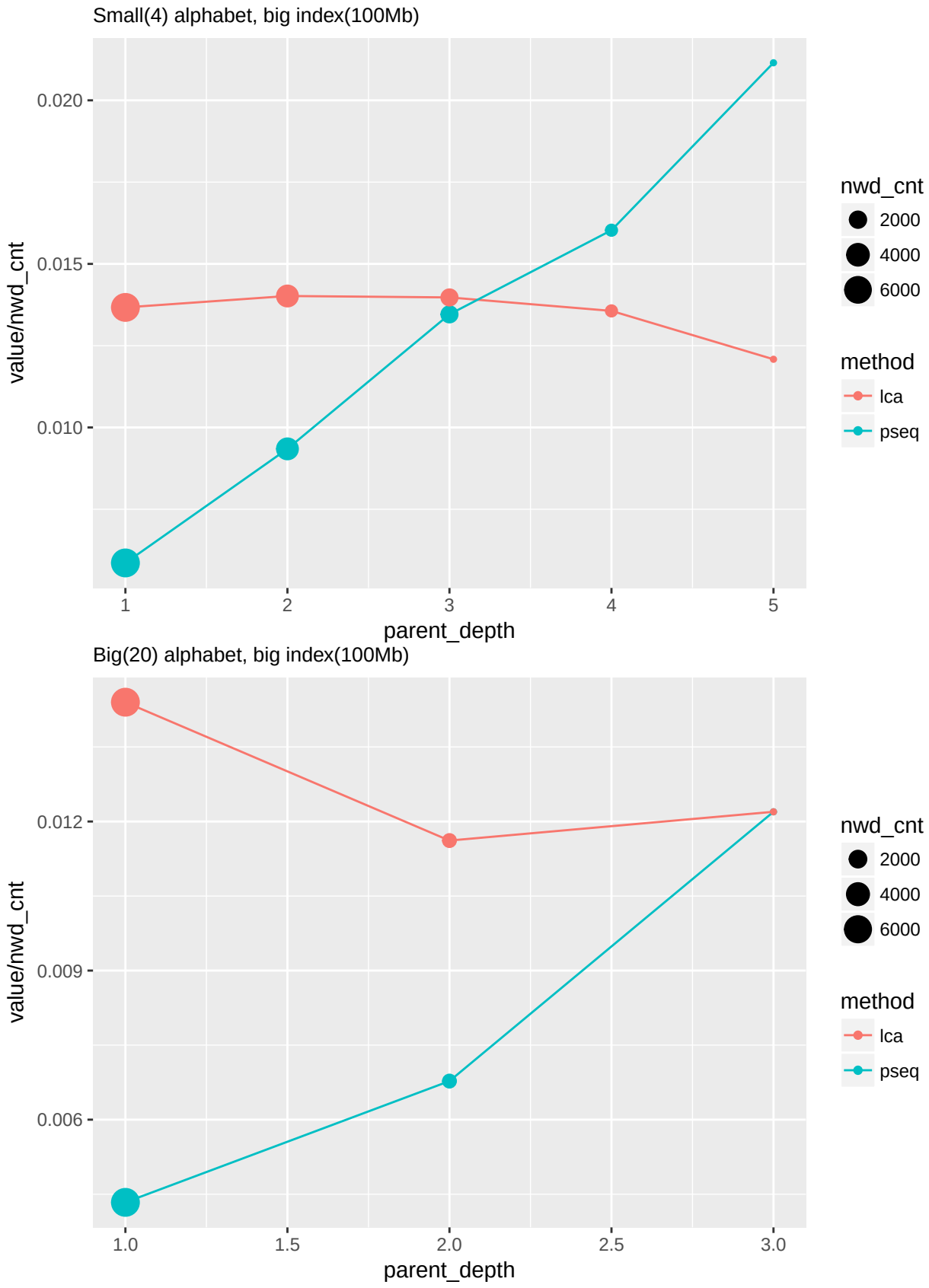
4.1.2 lowest maximal ancestor

Generate all tuples `(v, c, d)` from a tree where, `n` is an internal node, `c` a character, and `d = depth(v) - depth(u)` with `u` the lowest maximal ancestor.

Then shuffle and time the two ways of finding `u`: using the maxrep or with a sequence of parent calls.

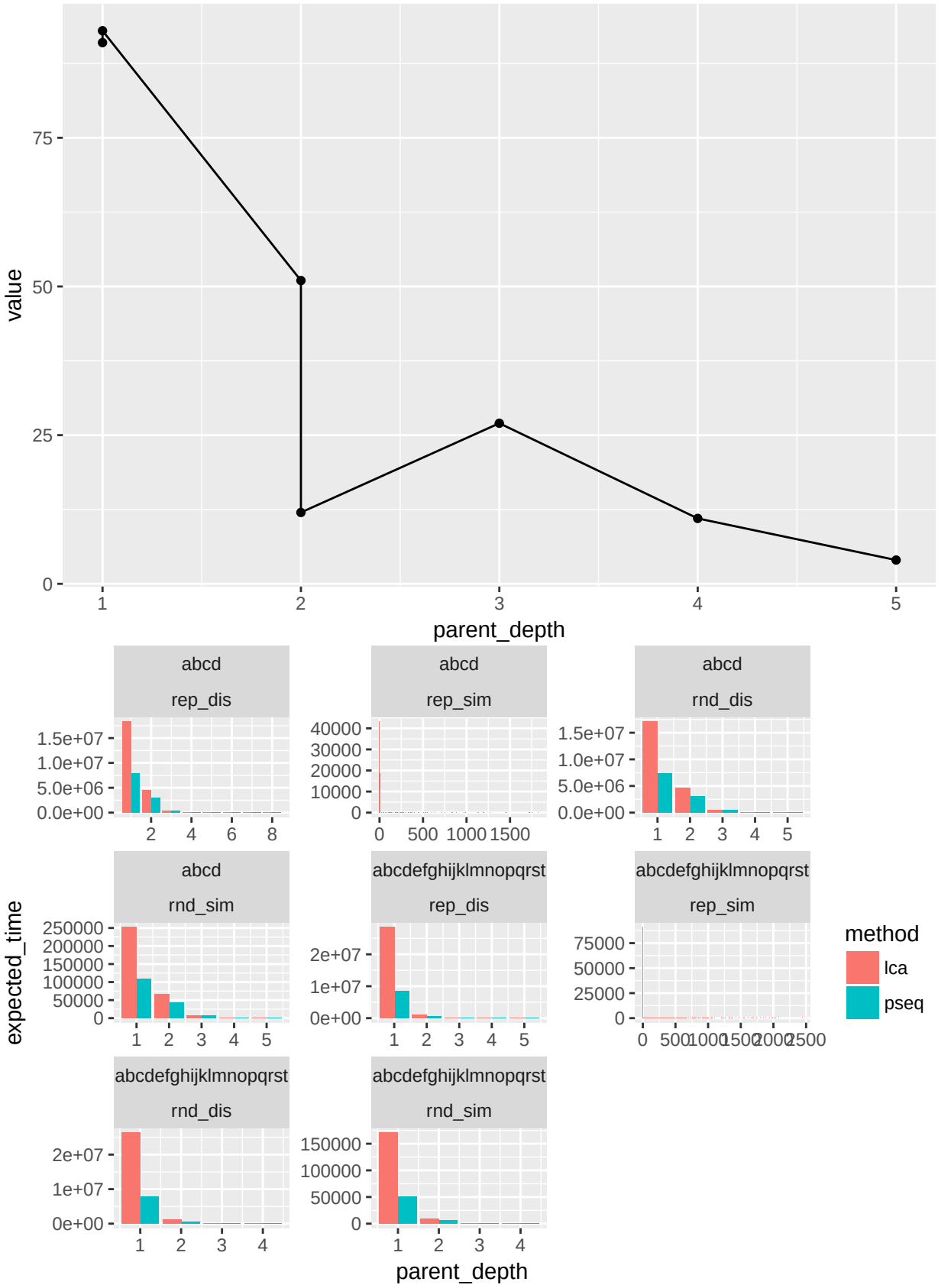
The plots show the `total time / nr of calls` for a given depth with the size of the point representing the `nr of calls` for a given depth value.

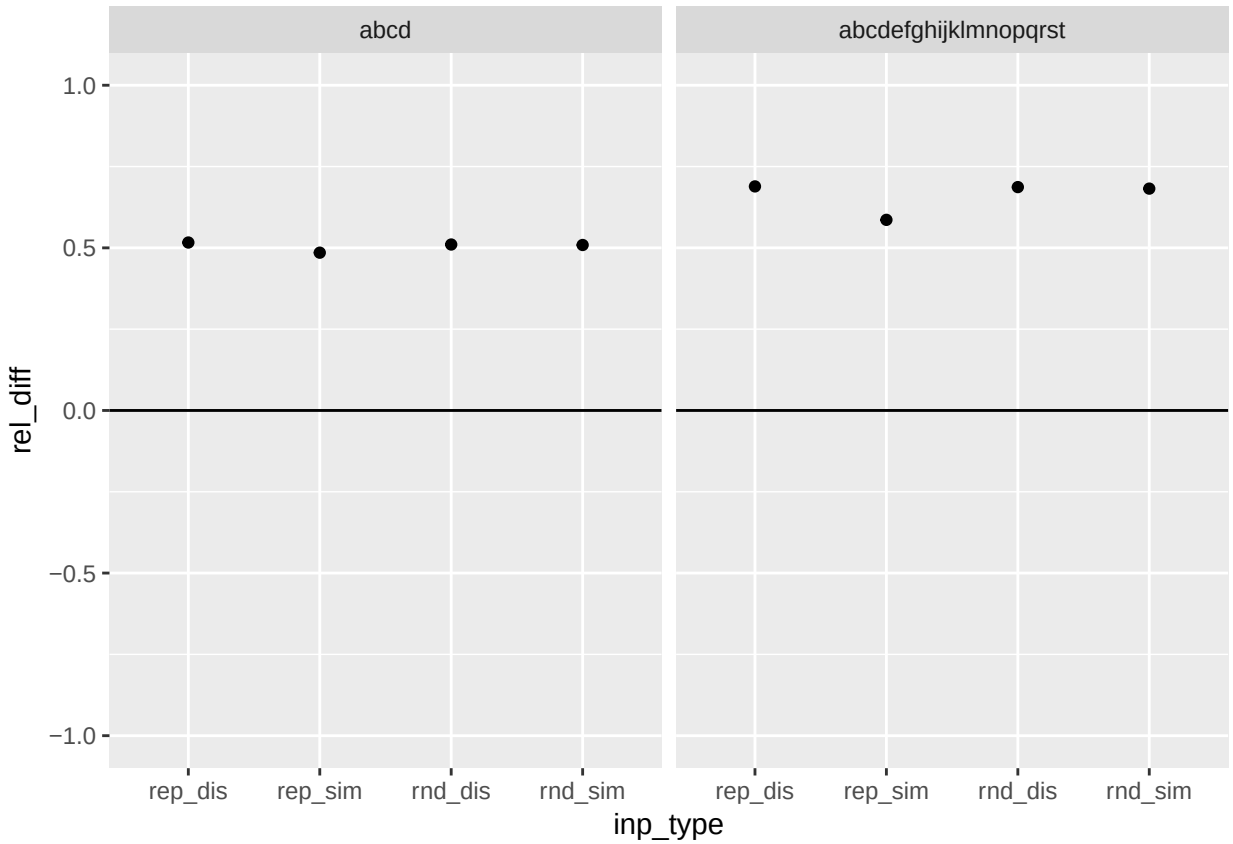
When the nr of calls is small (this happens for large depth lengths) the time resolution is not big enough and the numbers are not to be trusted – hence I have removed the data from the plots.



By counting the number of consecutive parent calls (for each number of consecutive calls) we project the

runtime of each method for an input type as below.





4.2 full tests

5 parallelization

Run the program on 1, 2, 4, 8 and 16 threads and measure the time it takes to build the RUNS and MS vectors.

> TODO: there is currently a bug on the parallel version of the program for particular inputs. At the moment fixing it is not a priority.