Softwarepraktikum SS 2019
Assignment 4

Group - 6

Roman Vuskov	376809	roman.vuskov@rwth-aachen.de
Songran Shi	357791	songran.shi@rwth-aachen.de
Marlene Damm	379945	marlene.damm@rwth-aachen.de
Marvin Gazibarić	378154	marvin.gazibaric@rwth-aachen.de

Task 1

In assignment 3, we already implemented a beam search algorithm. The moves are evaluated and the best k moves are sorted into a list we call the search beam while all other moves are discarded. k is therefore the *beam width. Move sorting*, in comparison, is thus equivalent to setting $k = \infty$, i.e. without discarding any moves.

Implementation-wise we could have used our beam search with beam width set to a very high value. But this would have resulted in a heavy performance penalty since the $\mathcal{O}(n \cdot k)$[1], algorithm we use to sort our beam would degenerate to complexity $\mathcal{O}(n^2)$. Instead we implemented a dedicated move sorting method `getOrderedMoves()` that uses a heavily optimized $\mathcal{O}(n \log n)$ sorting mechanism from the Java standard library.

Additionally, we also allowed the option of not sorting moves at all for completeness' sake, although this is highly discouraged as it renders alpha-beta pruning and the upcoming aspiration windows essentially useless.

[1] n is the number of moves in a given game state.

Task 3

For this exercise, we implemented two new classes, `IterationHeuristic` and `PancakeWatchdog` to help our AI realize iterative deepening.

`IterationHeuristic` is used in between iterations of depth searches to estimate whether there is sufficient time for one more step in depth. Every iteration the method `doIteration()` is called, returning `true` when iterative deepening should be proceeded.

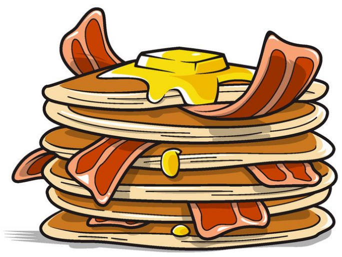
Currently the implementation is very basic. It aborts, i.e. returns `false`, if a certain time threshold is exceeded. That threshold is just $T \cdot h$, where T is the time limit for that particular move and h a hardcoded value, set to $h = 0.2$. We assume that search time scales linearly with number of nodes. Given that our beam width n is set to 5 per default, that value seems reasonable. This procedure works astonishingly well, with regard to its simplicity, but we still intend to implement more refined game-stage dependent time budget management in the future.

`PancakeWatchdog` is a safeguard against timeout disqualifications in case our iteration heuristic underestimated the time needed to complete the next iteration of depth search and gave it green light despite time shortage. The time check method `isPancake()` gets called frequently during the actual depth search, both in `evaluateNode()` and `getBeamMoves()`, to catch any impending timeout before it occurs with a safety margin currently set to $100ms$.

Nobody likes burnt pancakes, so be sure to turn off the heat 100 ms early!

To sum up, iterative deepening is realized by a `while`-loop in our AI class that calls `BRSNode` repeatedly with increasing depth limit so long as `IterationHeuristic` considers time budget abundant, and any false estimatates are caught by the

`PancakeWatchdog`. The best move is updated with the best move of each search iteration. In case a search is terminated prematurely, the best move of that iteration remains `null` and the result from previous iterations are used instead.



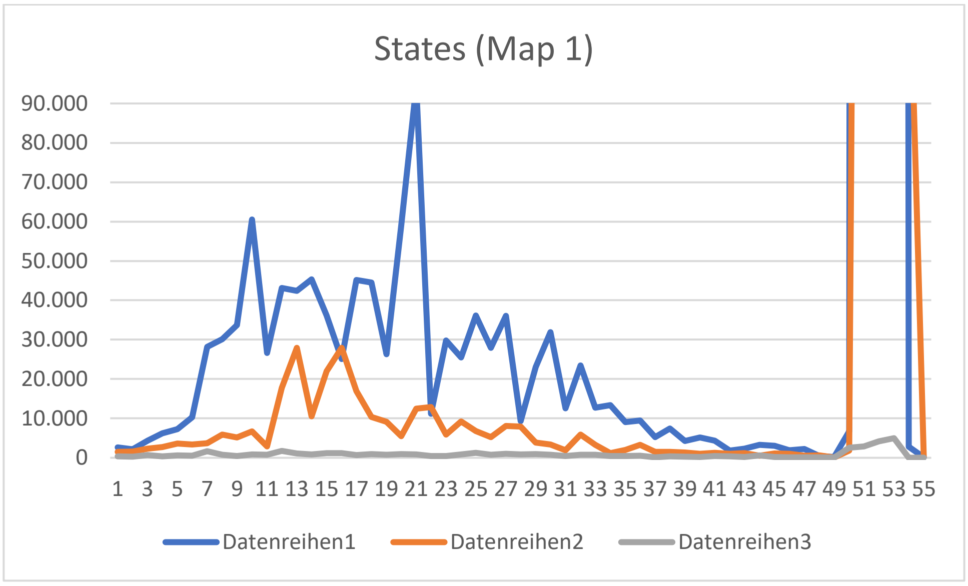
Bacon and pancakes, a timeless duo

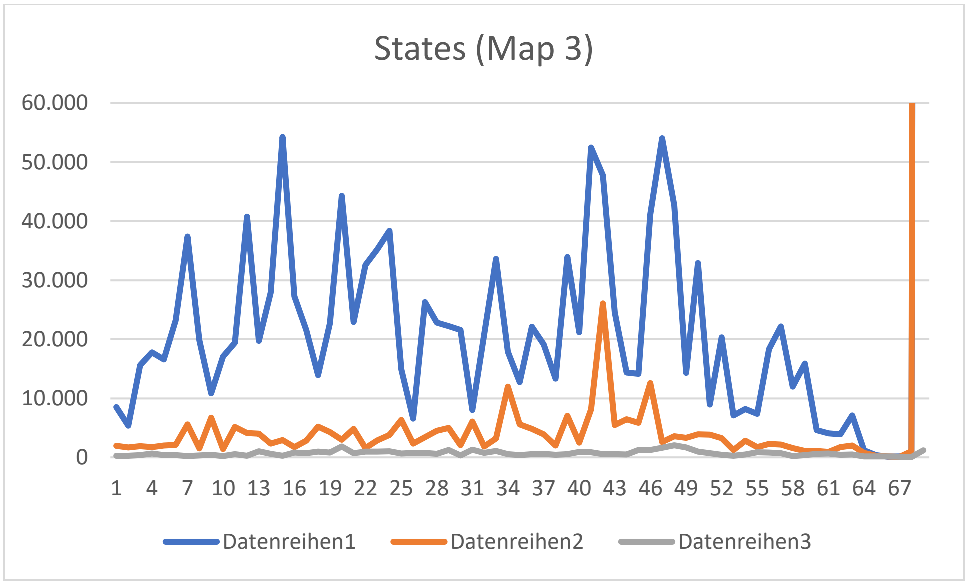
Task 2

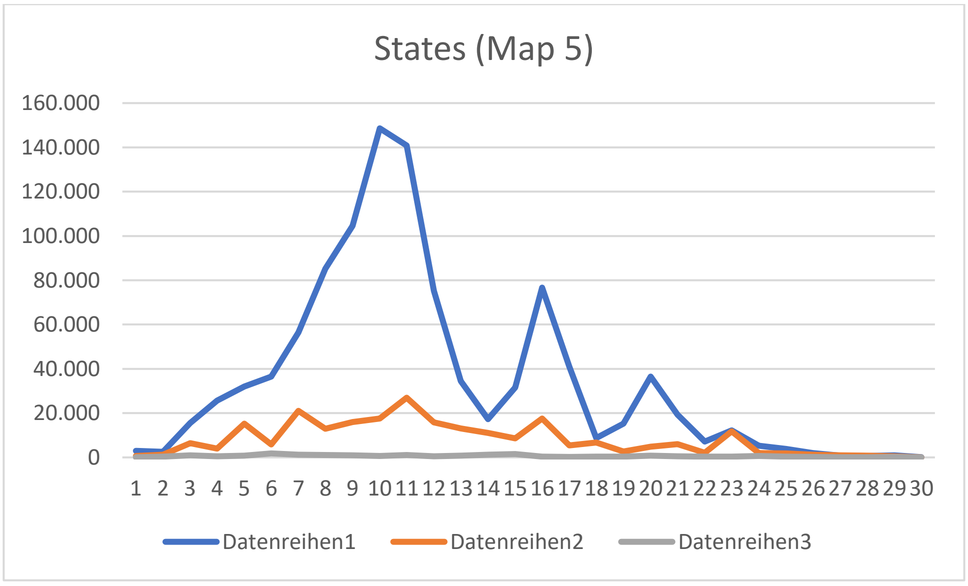
Below we provide graphs showing the time efficiency of constant depth searches with and without move sorting and beam search. It is apparent that time per move is highly correlated with number of states evaluated. As expected, sorting moves according to heuristic values results in a moderate speedup (3-10x) compared with no-move-ordering searches due to more pruning opportunities. This, conversely, gives us confidence that our board state rating heuristics are at least self-consistent, i.e. board states with high heuristic values tend to produce child nodes also with high heuristic values.

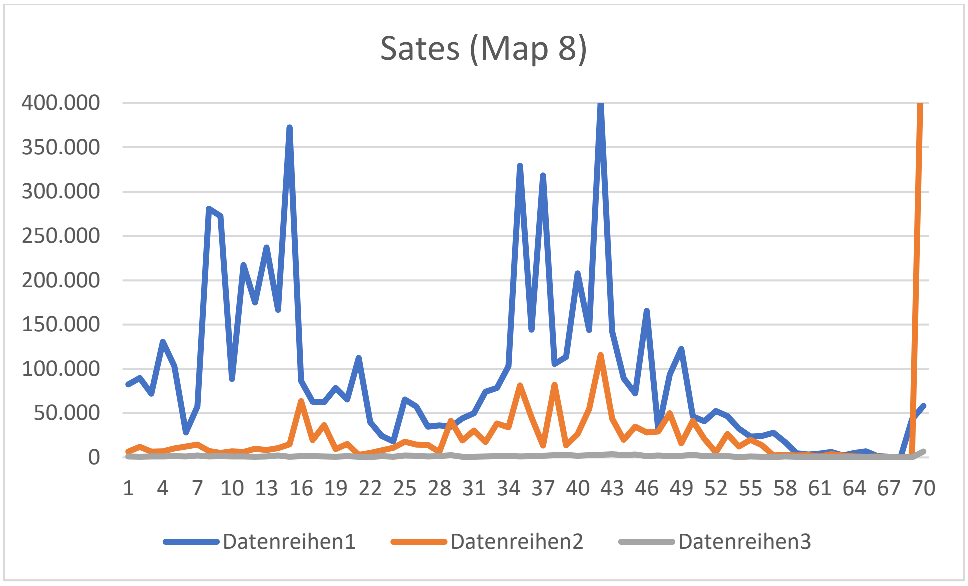
Of course, beam search produces an even greater speedup (up to 100x) compared with no-move-ordering searches. More importantly, with beam search time per move remains nearly constant throughout the game, while without beam search time per move jumps wildly depending on how many moves are available in a particular board configuration (generally starts out low in early game and reaches local maximum in mid- to late-game, then explodes in "override phase"). Practically speaking, we would have to continue using beam search or implement far faster heuristics in order to not exceed the standard 1 second time limit.

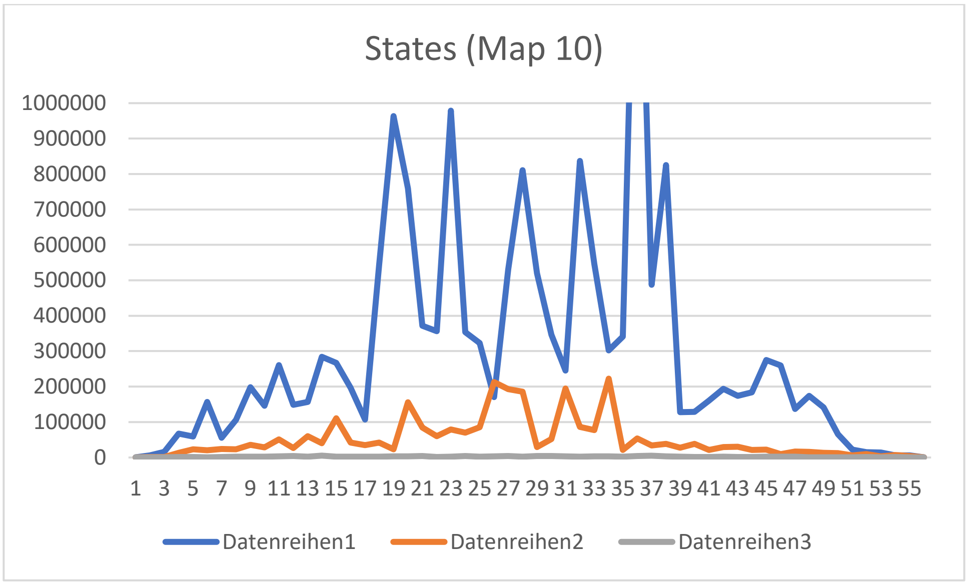
States per move for set depth of 4.
Datenreihe1: no sort (blue)
Datenreihe2: with sort (orange)
Detenreihe3: with sort and beam of 5 (gray)

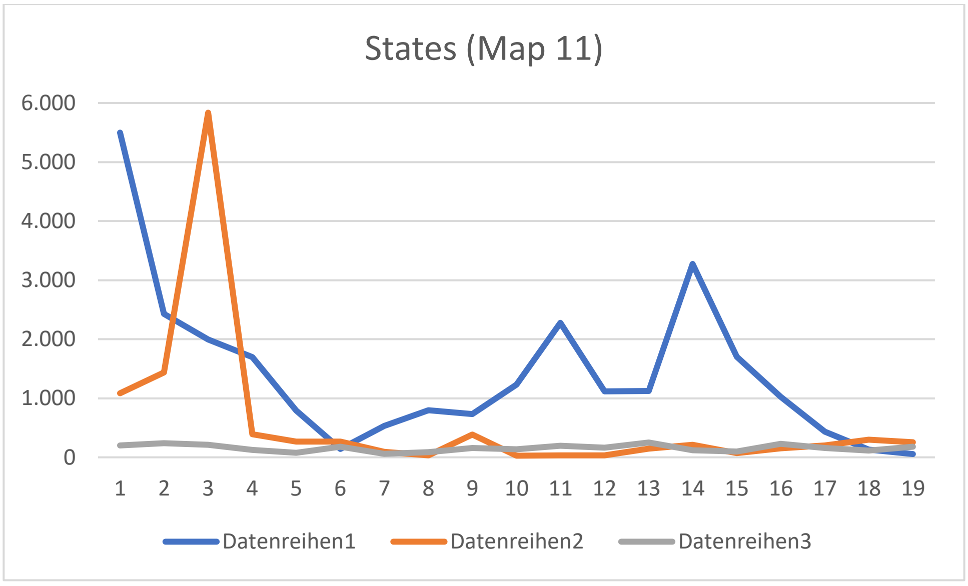


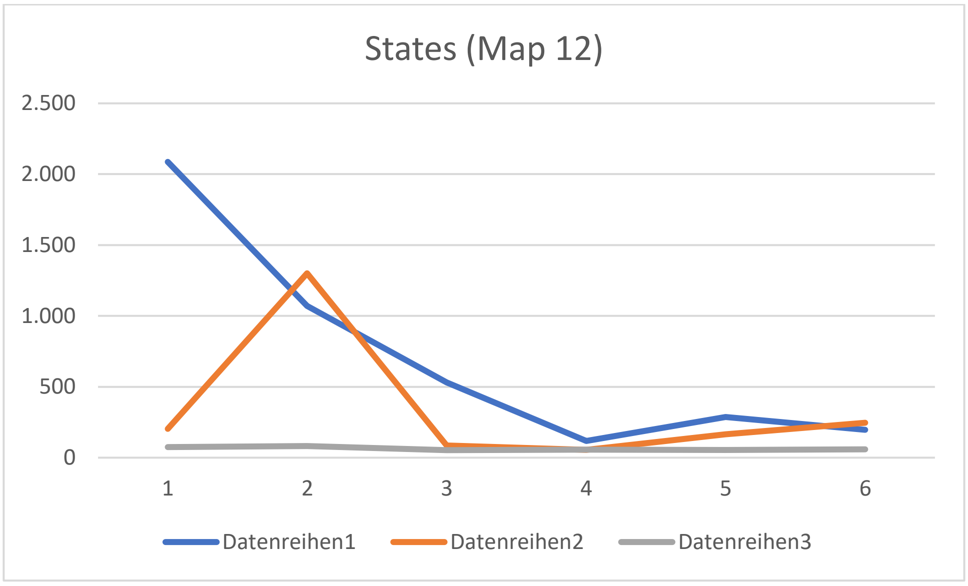


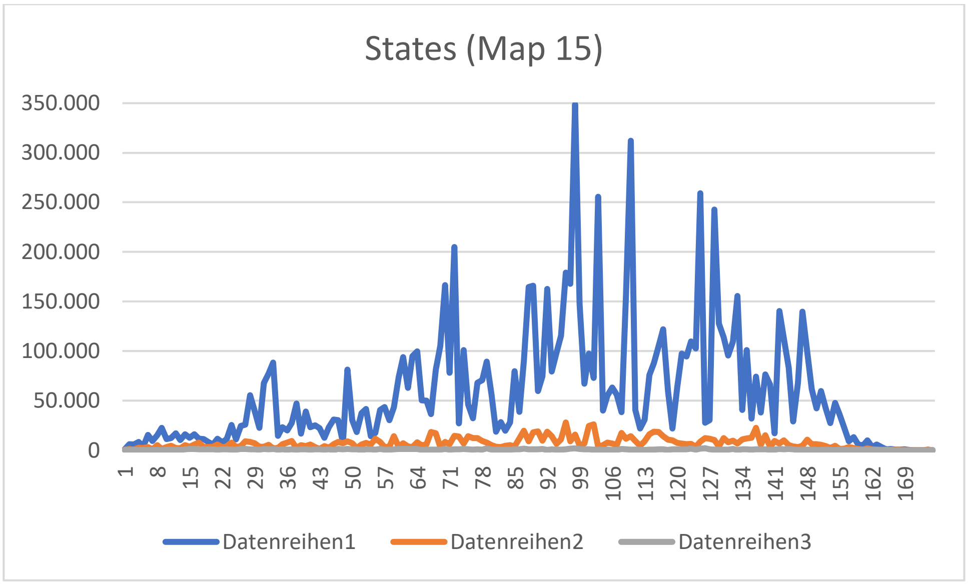


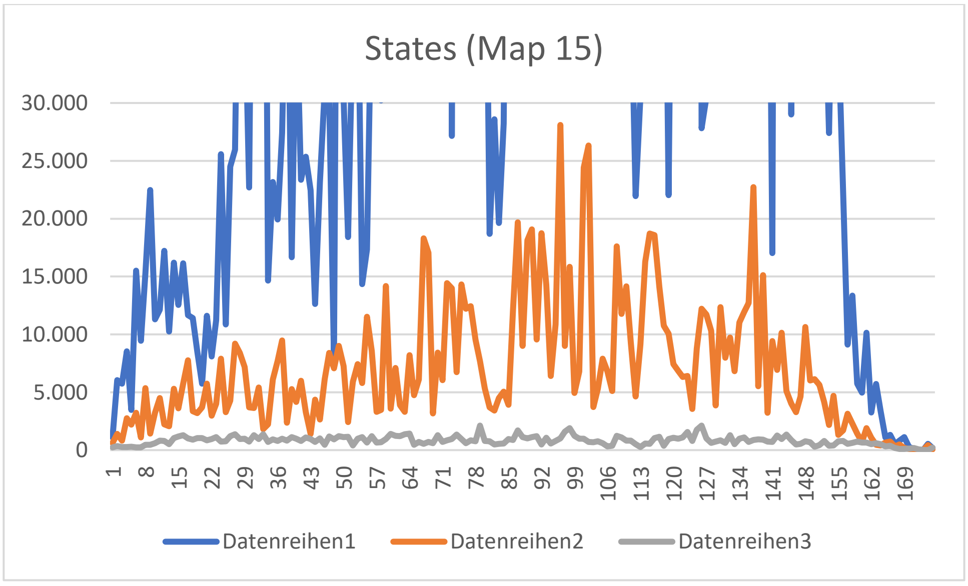

Sates (Map 8)
400.000
350.000
300.000
250.000
200.000
150.000
100.000
50.000
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70
Datenreihen1
Datenreihen2
Datenreihen3

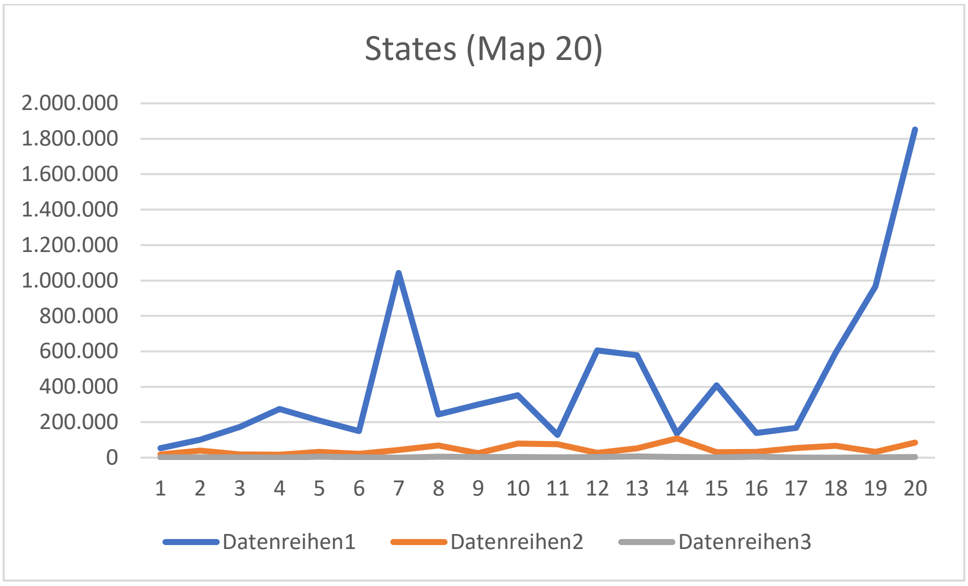

States (Map 10)
1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55
Datenreihen1
Datenreihen2
Datenreihen3


States (Map 11)
6.000
5.000
4.000
3.000
2.000
1.000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Datenreihen1
Datenreihen2
Datenreihen3

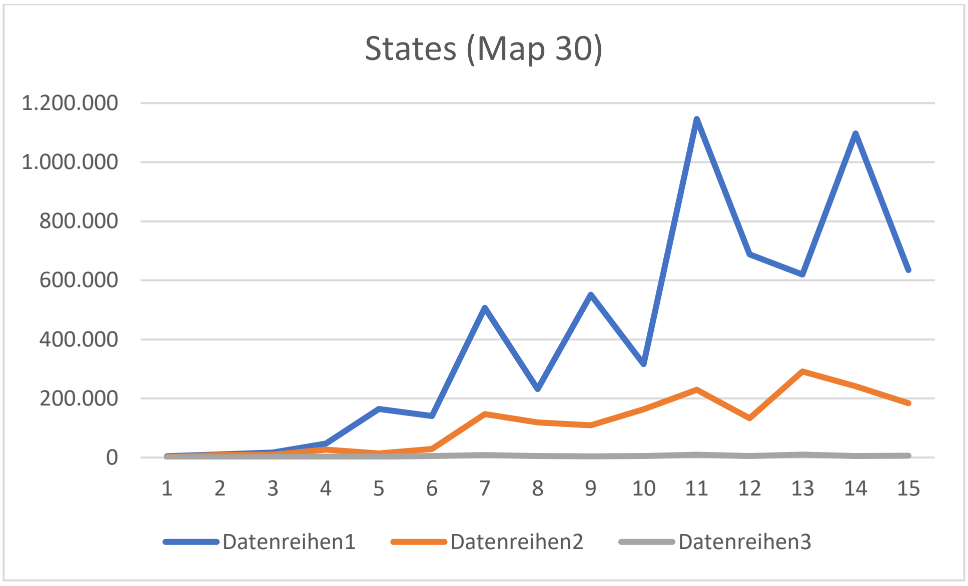

States (Map 12)
2.500
2.000
1.500
1.000
500
0
1
2
3
4
5
6
Datenreihen1
Datenreihen2
Datenreihen3


States (Map 15)
350.000
300.000
250.000
200.000
150.000
100.000
50.000
0
1
8
15
22
29
36
43
50
57
64
71
78
85
92
99
106
113
120
127
134
141
148
155
162
169
Datenreihen1
Datenreihen2
Datenreihen3

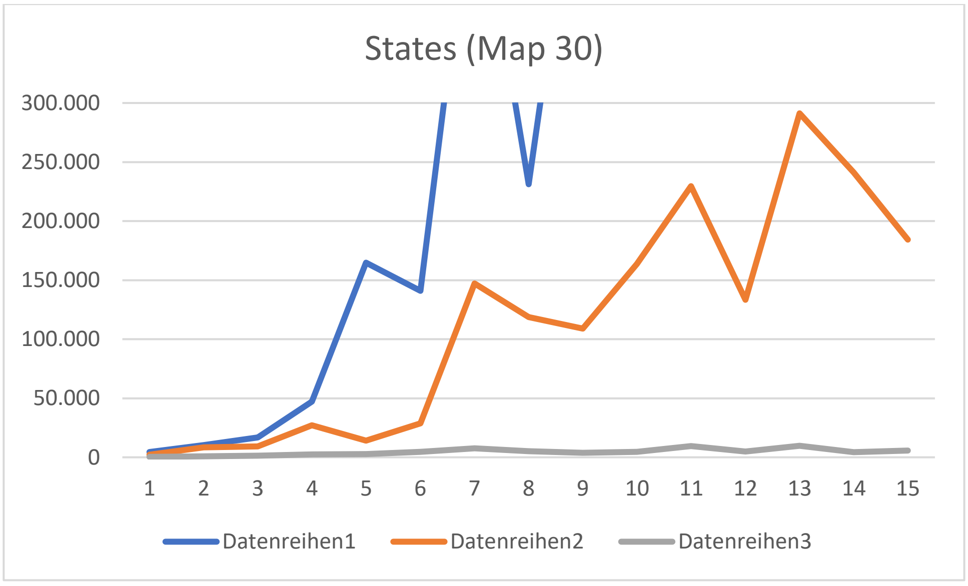

States (Map 15)
30.000
25.000
20.000
15.000
10.000
5.000
0
1
8
15
22
29
36
43
50
57
64
71
78
85
92
99
106
113
120
127
134
141
148
155
162
169
Datenreihen1
Datenreihen2
Datenreihen3



States (Map 20)
2.000.000
1.800.000
1.600.000
1.400.000
1.200.000
1.000.000
800.000
600.000
400.000
200.000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Datenreihen1
Datenreihen2
Datenreihen3

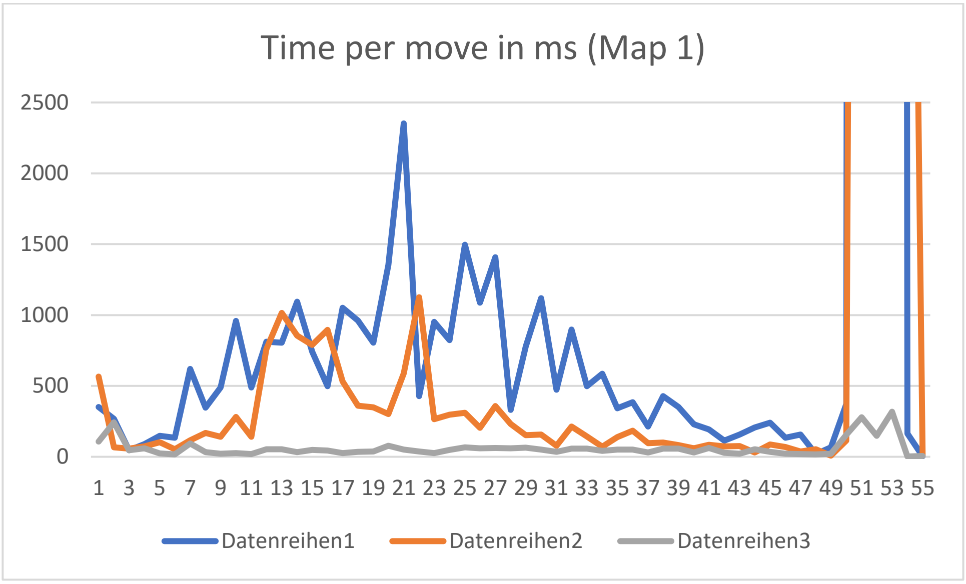


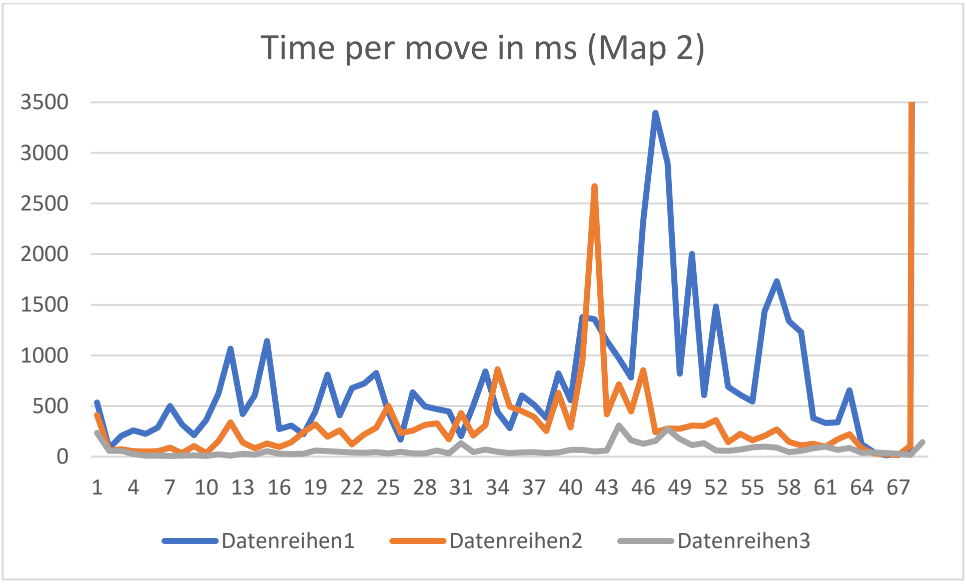
States (Map 30)
1.200.000
1.000.000
800.000
600.000
400.000
200.000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Datenreihen1
Datenreihen2
Datenreihen3

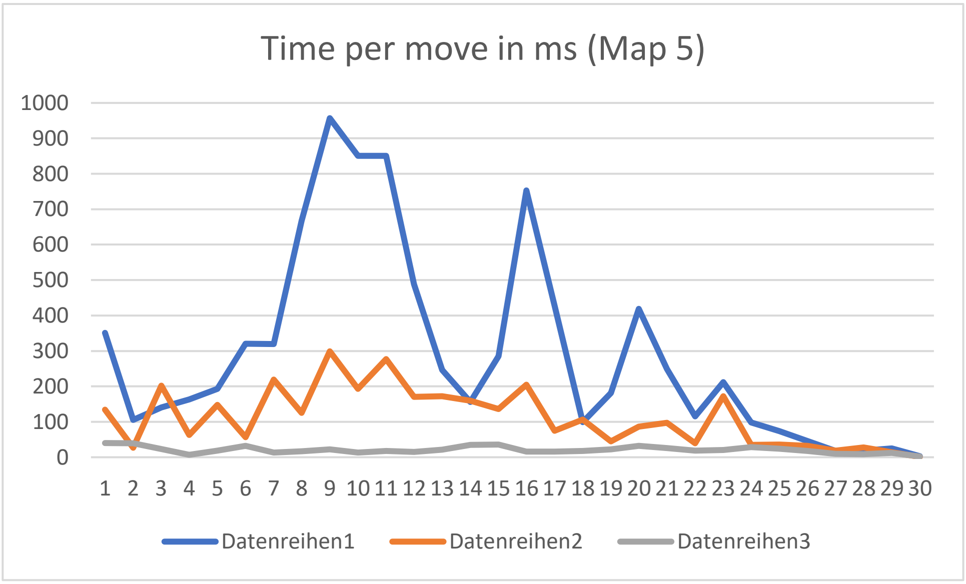


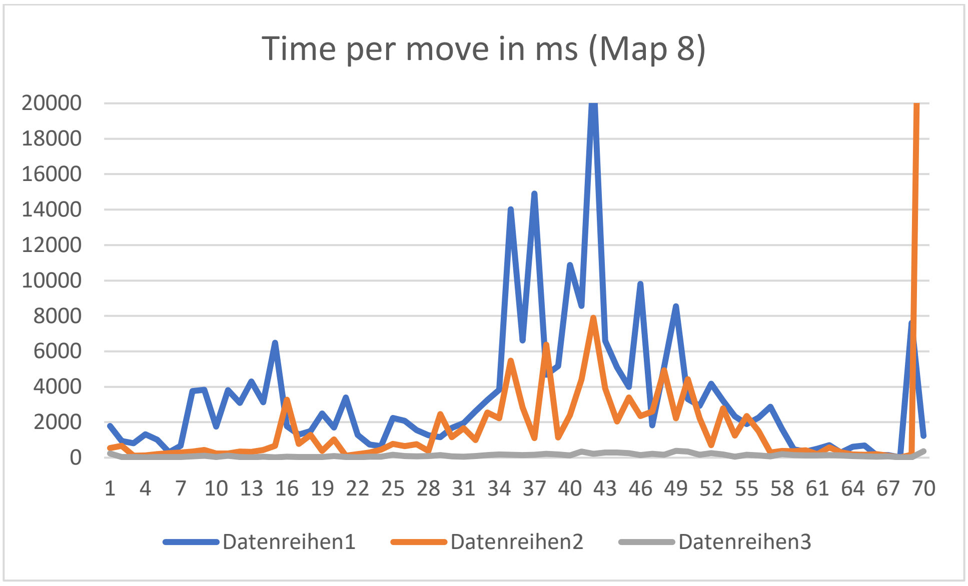
States (Map 30)
300.000
250.000
200.000
150.000
100.000
50.000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Datenreihen1
Datenreihen2
Datenreihen3

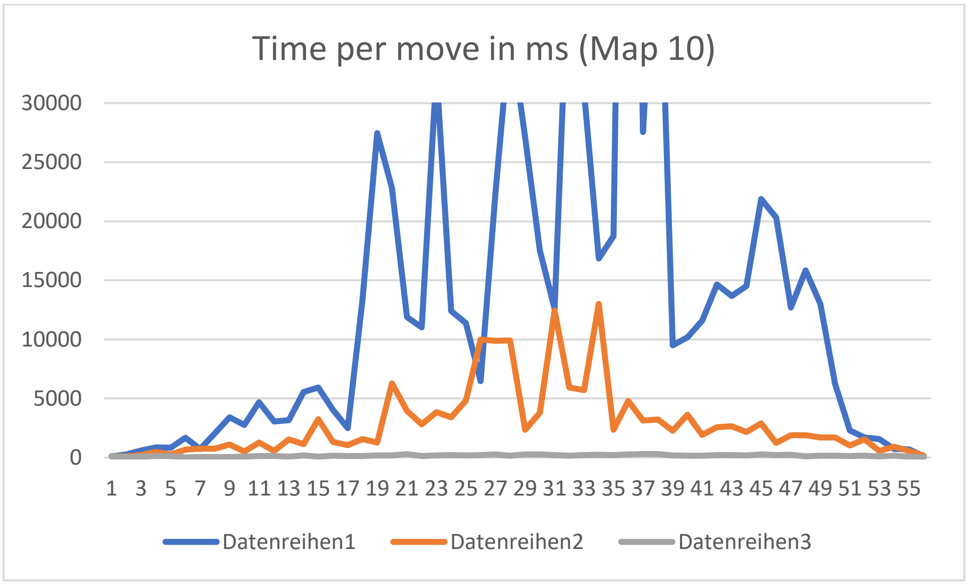
Times per move for set depth of 4.
Datenreihe1: no sort (blue)
Datenreihe2: with sort (orange)
Detenreihe3: with sort and beam of 5 (gray)

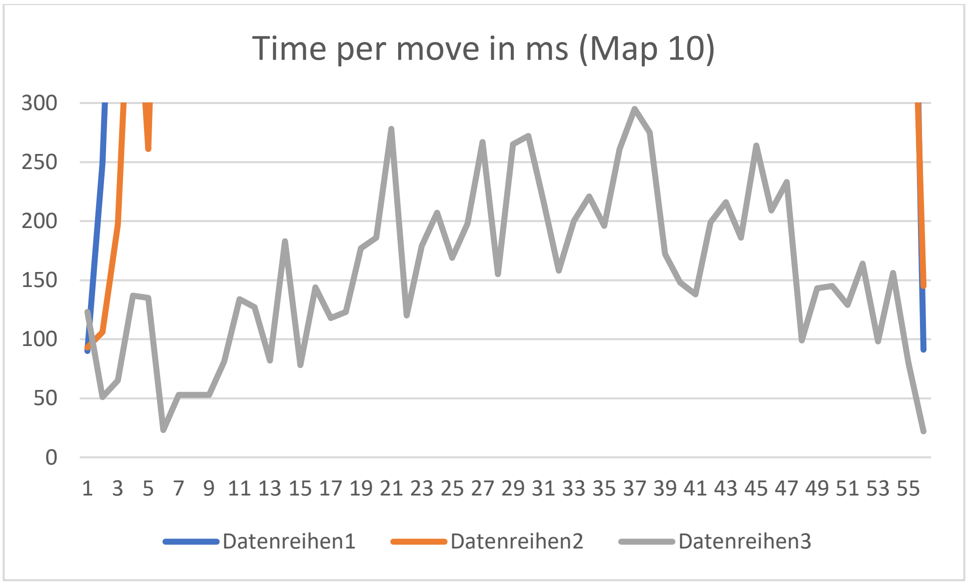


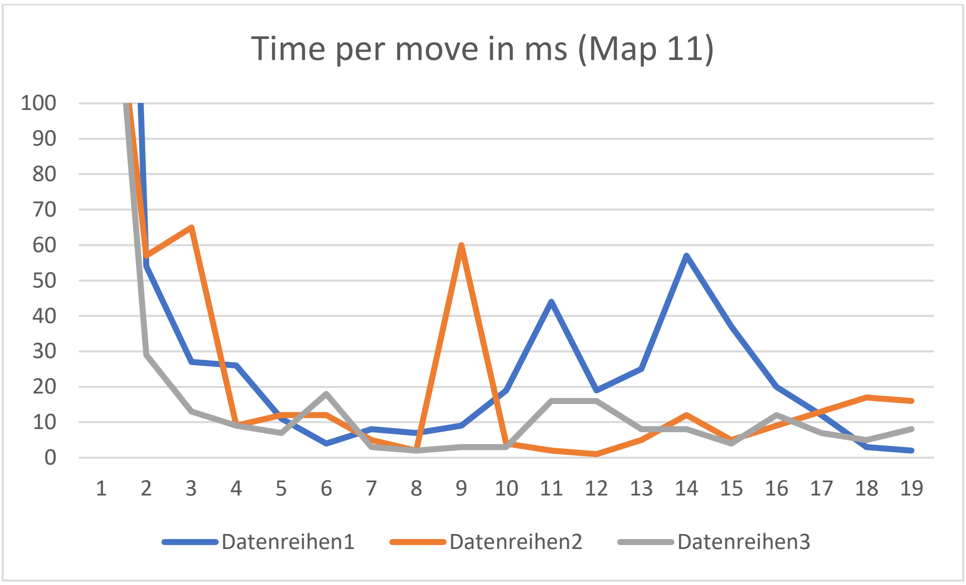


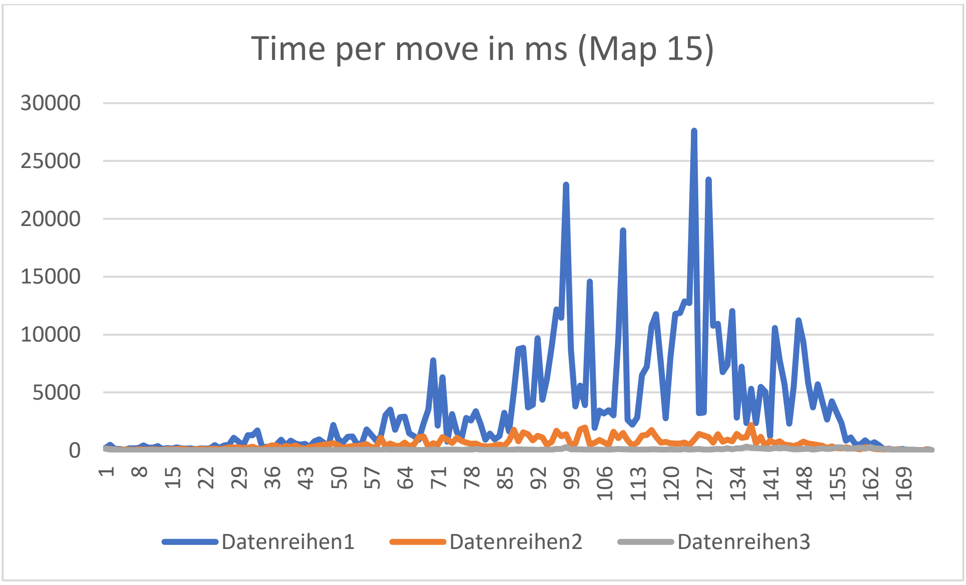


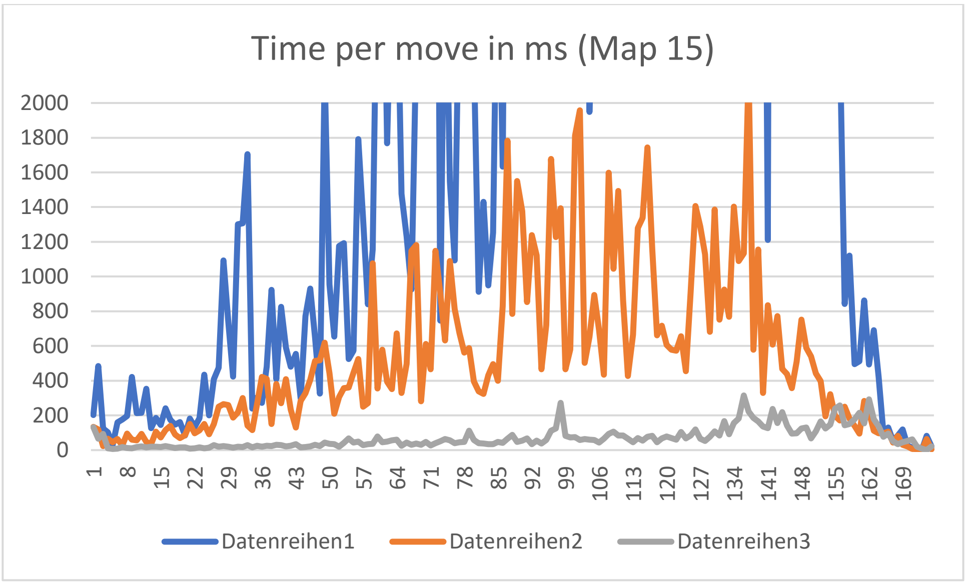

Time per move in ms (Map 8)
20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70
Datenreihen1
Datenreihen2
Datenreihen3

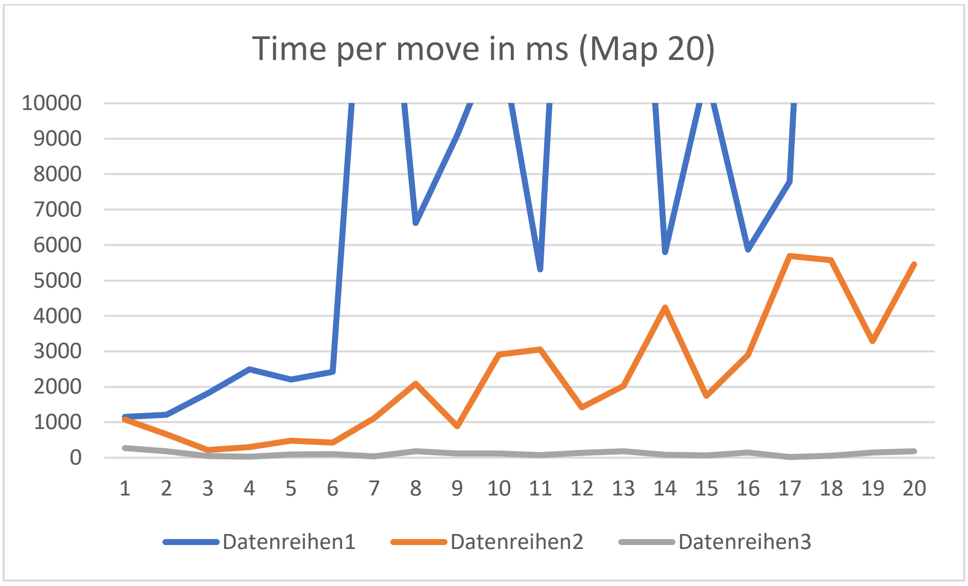

Time per move in ms (Map 10)
30000
25000
20000
15000
10000
5000
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55
Datenreihen1
Datenreihen2
Datenreihen3

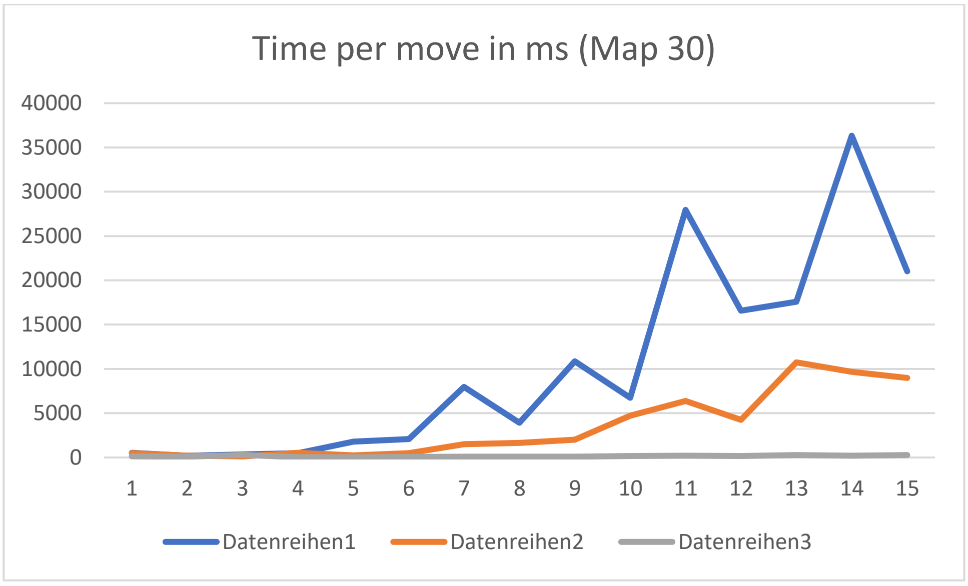

Time per move in ms (Map 10)
300
250
200
150
100
50
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55
Datenreihen1
Datenreihen2
Datenreihen3

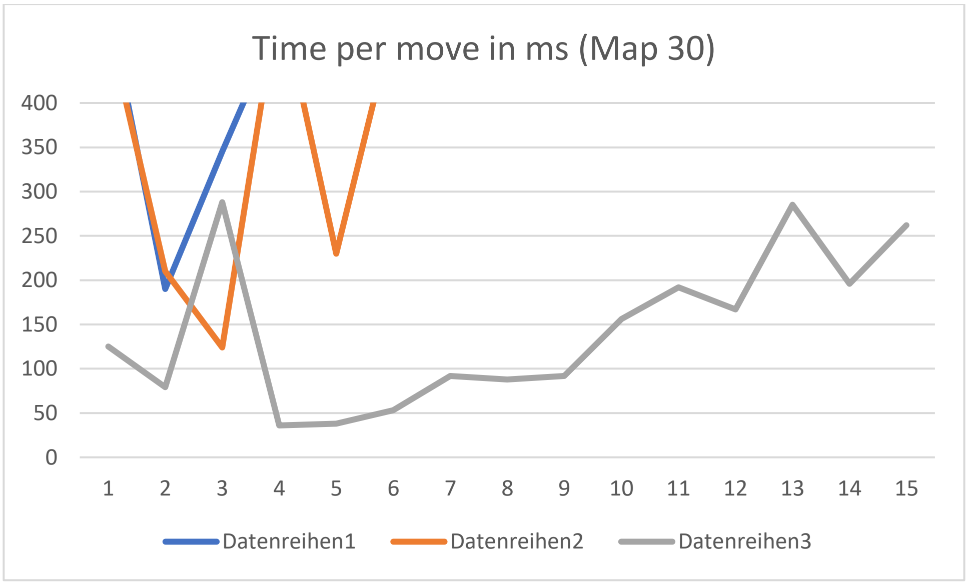

Time per move in ms (Map 11)
100
90
80
70
60
50
40
30
20
10
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Datenreihen1
Datenreihen2
Datenreihen3


Time per move in ms (Map 15)
30000
25000
20000
15000
10000
5000
0
1 8 15 22 29 36 43 50 57 64 71 78 85 92 99 106 113 120 127 134 141 148 155 162 169
Datenreihen1
Datenreihen2
Datenreihen3

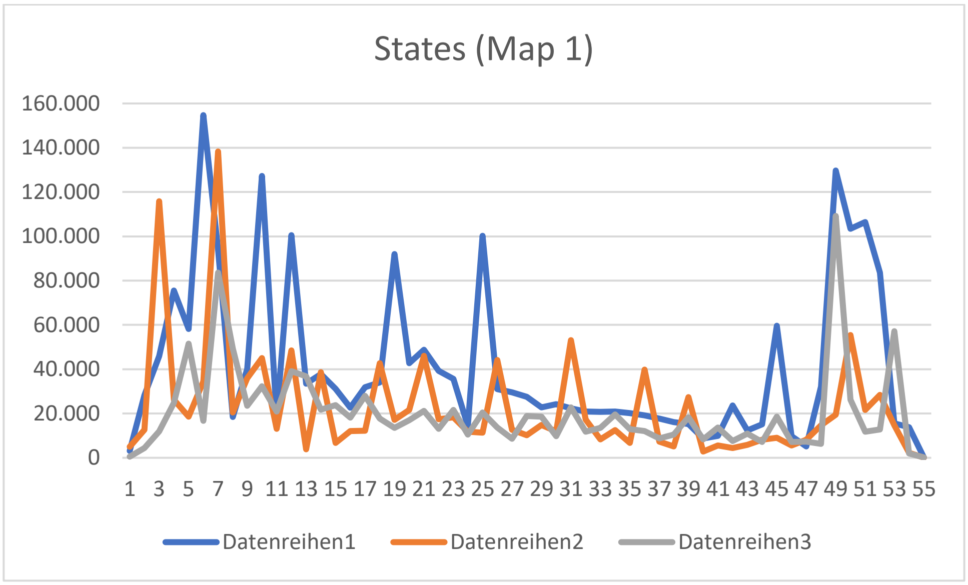

Time per move in ms (Map 15)
2000
1800
1600
1400
1200
1000
800
600
400
200
0
1 8 15 22 29 36 43 50 57 64 71 78 85 92 99 106 113 120 127 134 141 148 155 162 169
Datenreihen1
Datenreihen2
Datenreihen3

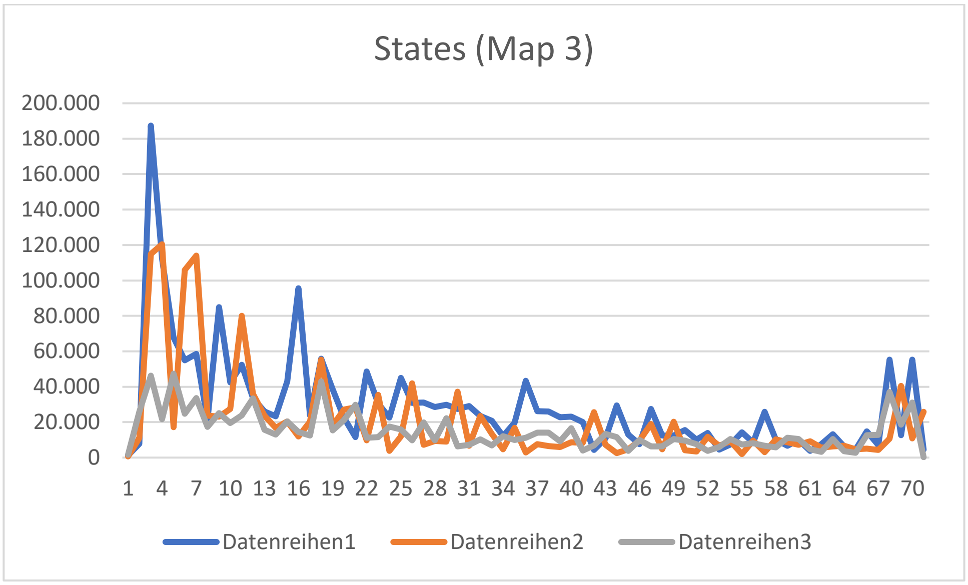

Time per move in ms (Map 20)
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Datenreihen1
Datenreihen2
Datenreihen3

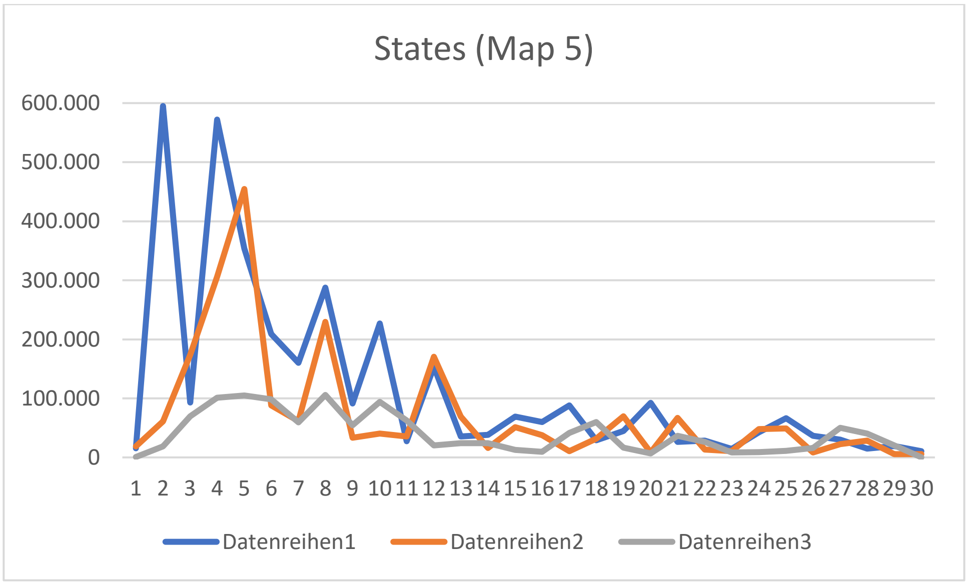

Time per move in ms (Map 30)
40000
35000
30000
25000
20000
15000
10000
5000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Datenreihen1
Datenreihen2
Datenreihen3

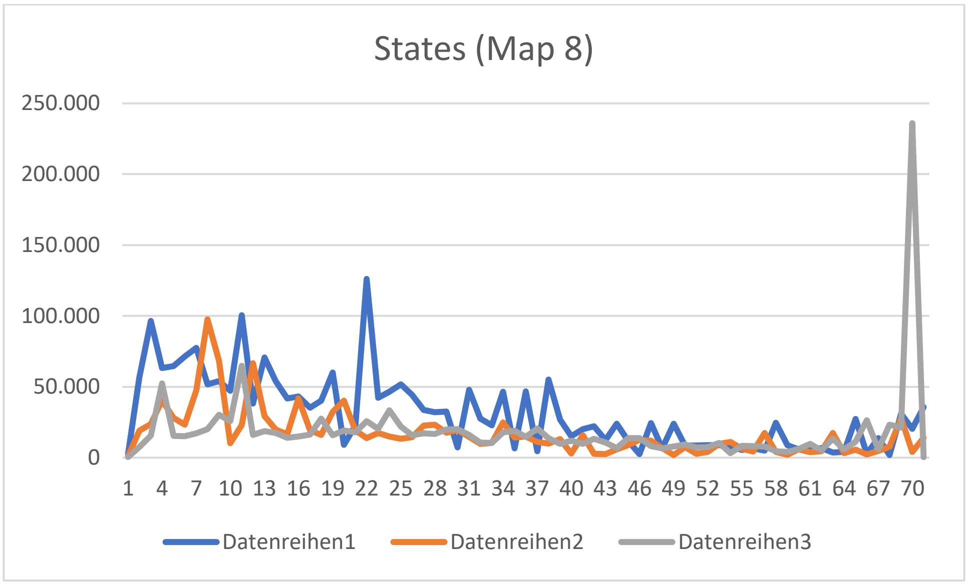

Time per move in ms (Map 30)
400
350
300
250
200
150
100
50
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Datenreihen1
Datenreihen2
Datenreihen3

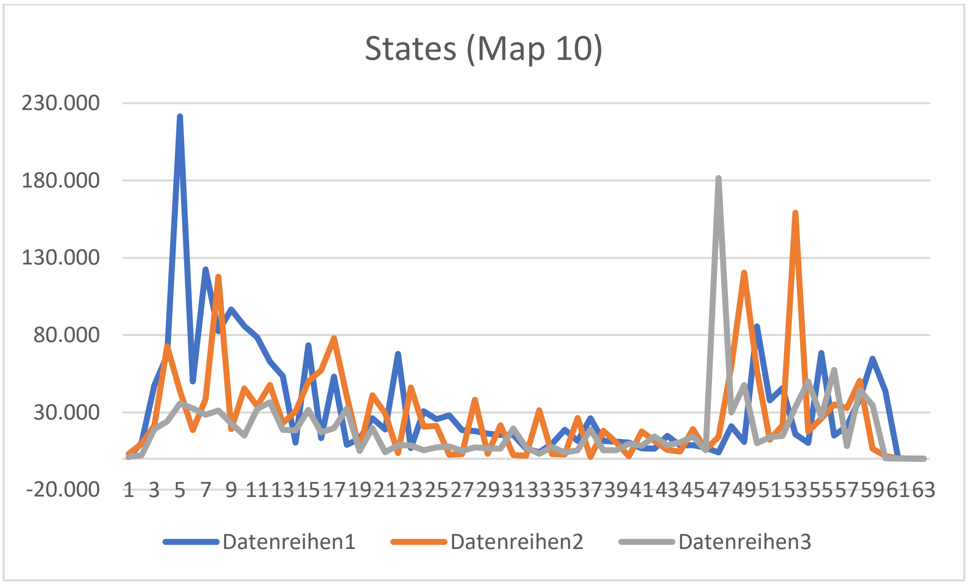
States per move for set time limit of 1s.
Datenreihe1: no sort (blue)
Datenreihe2: with sort (orange)
Detenreihe3: with sort and beam of 5 (gray)

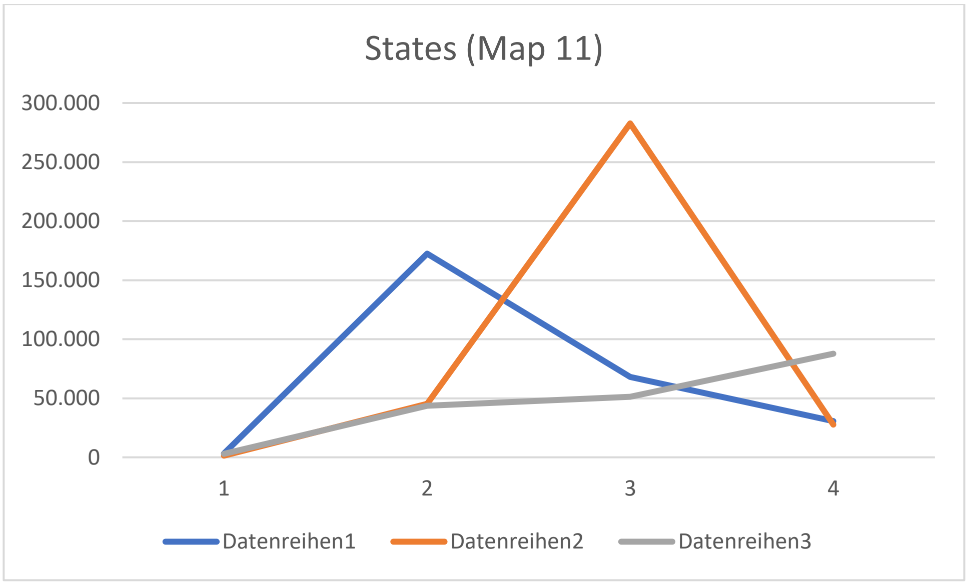


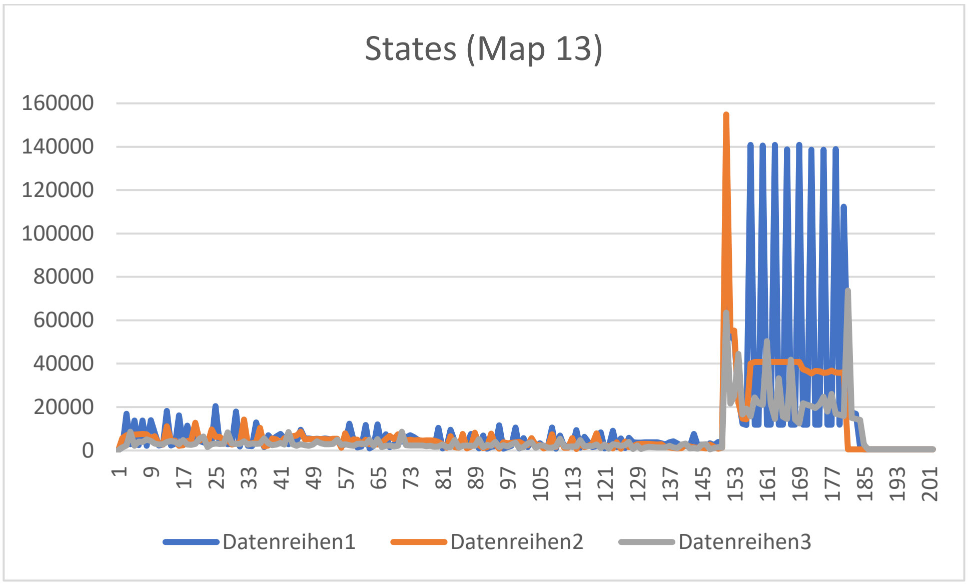


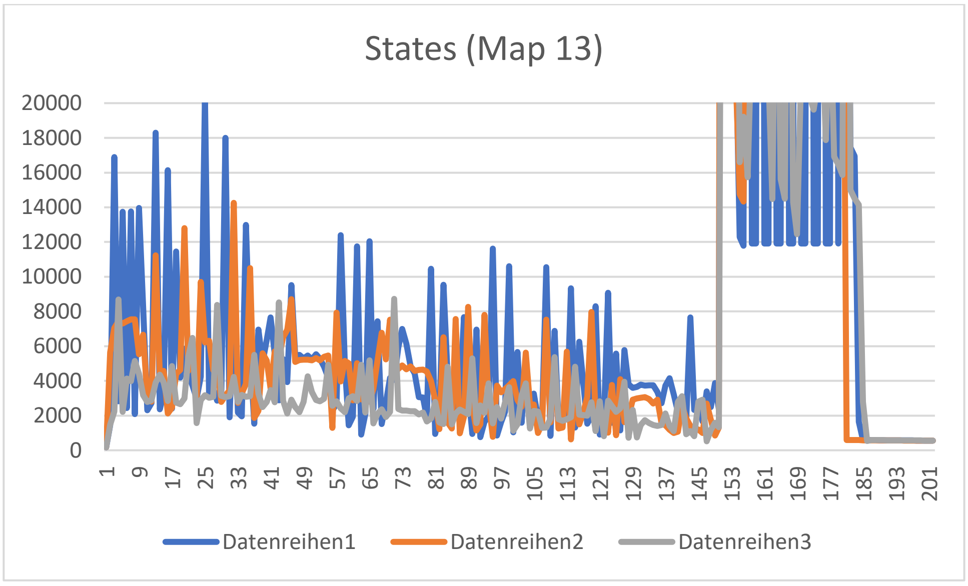


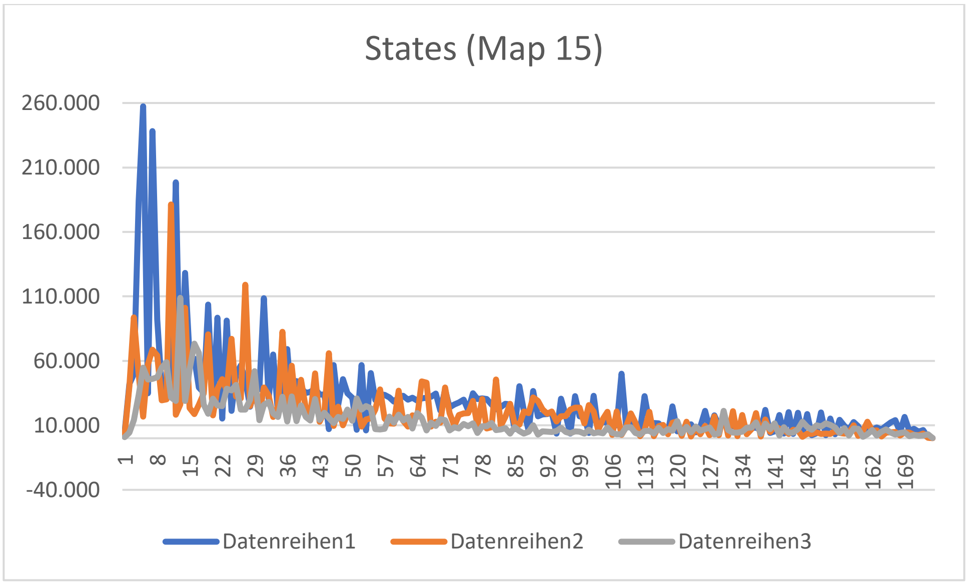

States (Map 8)
250.000
200.000
150.000
100.000
50.000
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70
Datenreihen1
Datenreihen2
Datenreihen3

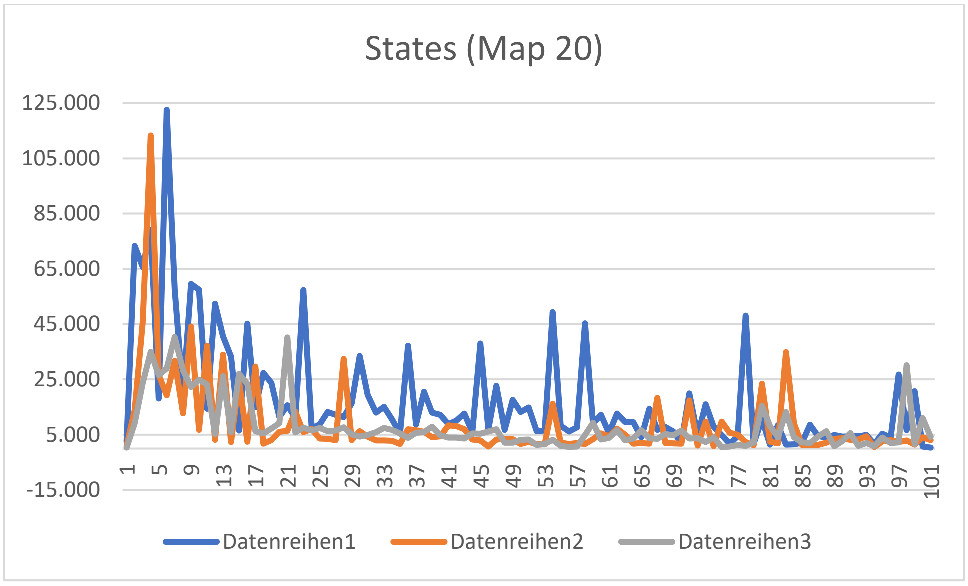

States (Map 10)
230.000
180.000
130.000
80.000
30.000
-20.000
1 3 5 7 9 11131517192123252729313335373941434547495153555759616 3
Datenreihen1
Datenreihen2
Datenreihen3

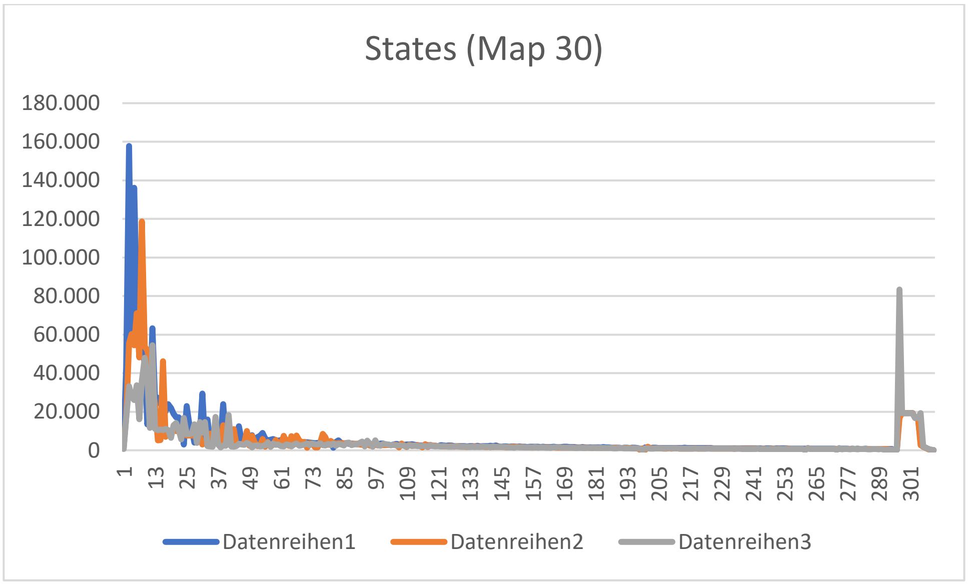

States (Map 11)
300.000
250.000
200.000
150.000
100.000
50.000
0
1
2
3
4
Datenreihen1
Datenreihen2
Datenreihen3

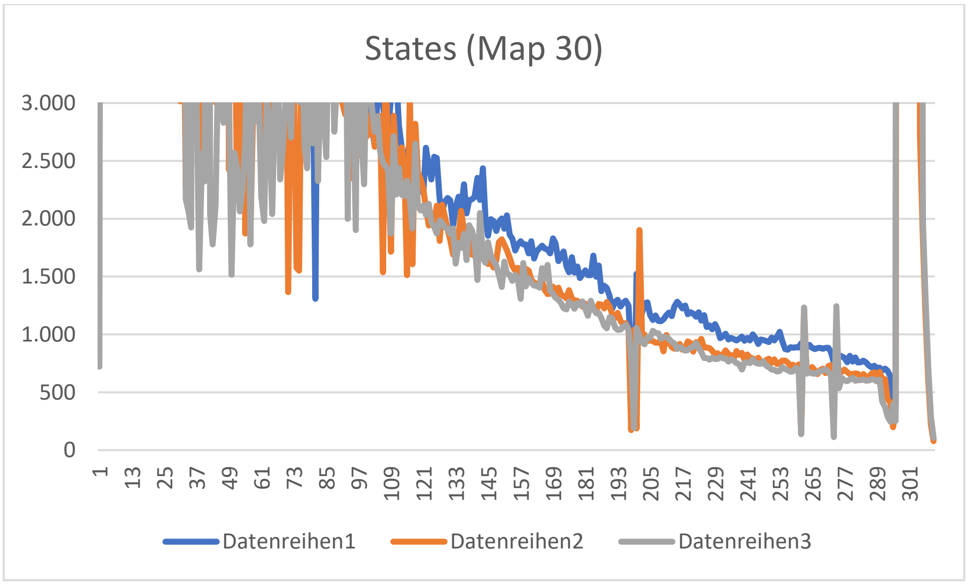

States (Map 13)
160000
140000
120000
100000
80000
60000
40000
20000
0
1
9
17
25
33
41
49
57
65
73
81
89
97
105
113
121
129
137
145
153
161
169
177
185
193
201
Datenreihen1
Datenreihen2
Datenreihen3


States (Map 13)
20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
1
9
17
25
33
41
49
57
65
73
81
89
97
105
113
121
129
137
145
153
161
169
177
185
193
201
Datenreihen1
Datenreihen2
Datenreihen3

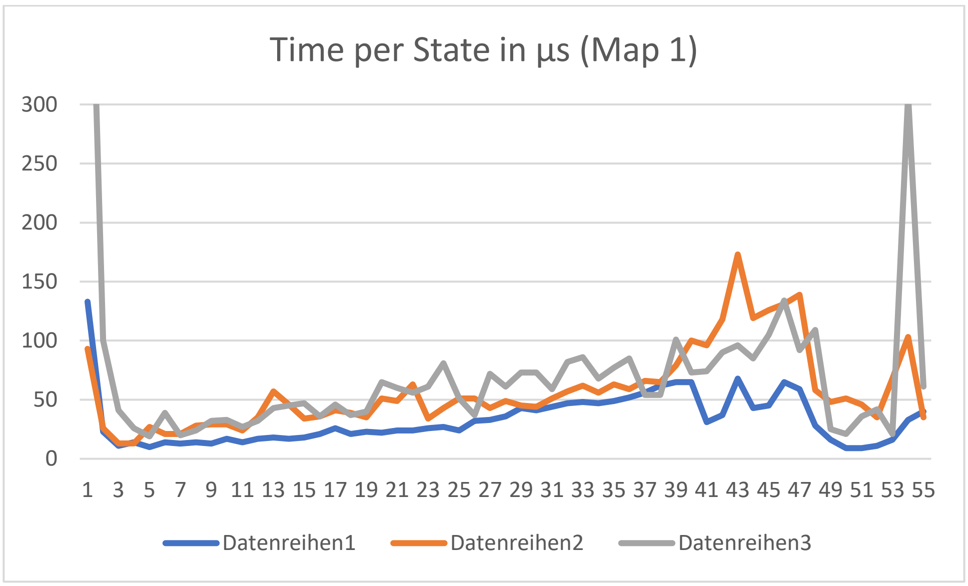

States (Map 15)
260.000
210.000
160.000
110.000
60.000
10.000
-40.000
1
8
15
22
29
36
43
50
57
64
71
78
85
92
99
106
113
120
127
134
141
148
155
162
169
Datenreihen1
Datenreihen2
Datenreihen3

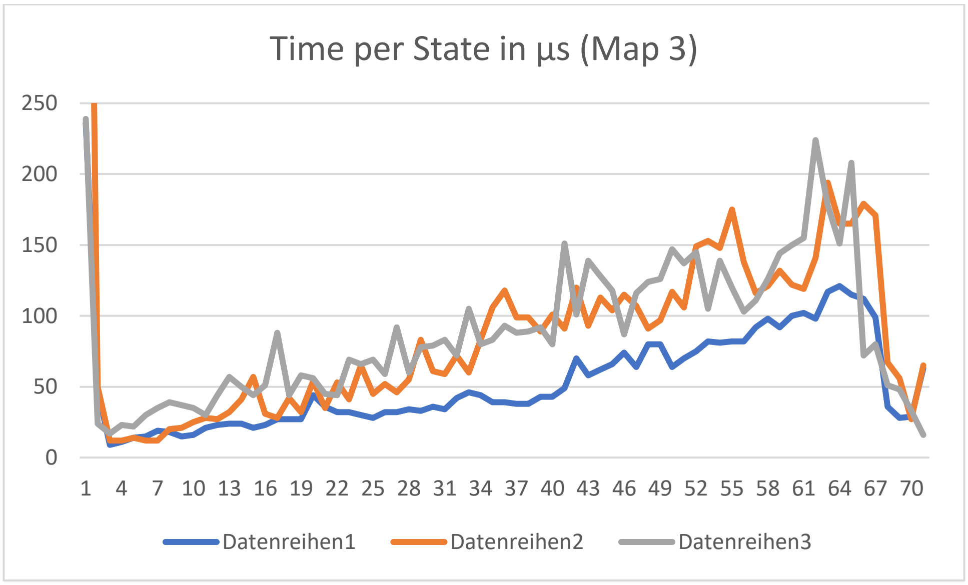

States (Map 20)
125.000
105.000
85.000
65.000
45.000
25.000
5.000
-15.000
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57
61
65
69
73
77
81
85
89
93
97
101
Datenreihen1
Datenreihen2
Datenreihen3

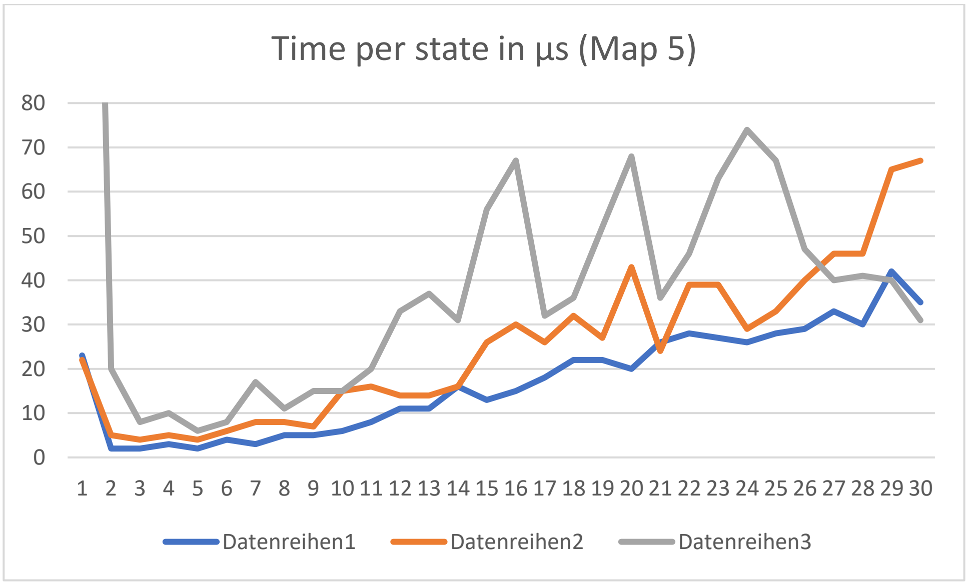

States (Map 30)
180.000
160.000
140.000
120.000
100.000
80.000
60.000
40.000
20.000
0
1
13
25
37
49
61
73
85
97
109
121
133
145
157
169
181
193
205
217
229
241
253
265
277
289
301
Datenreihen1
Datenreihen2
Datenreihen3

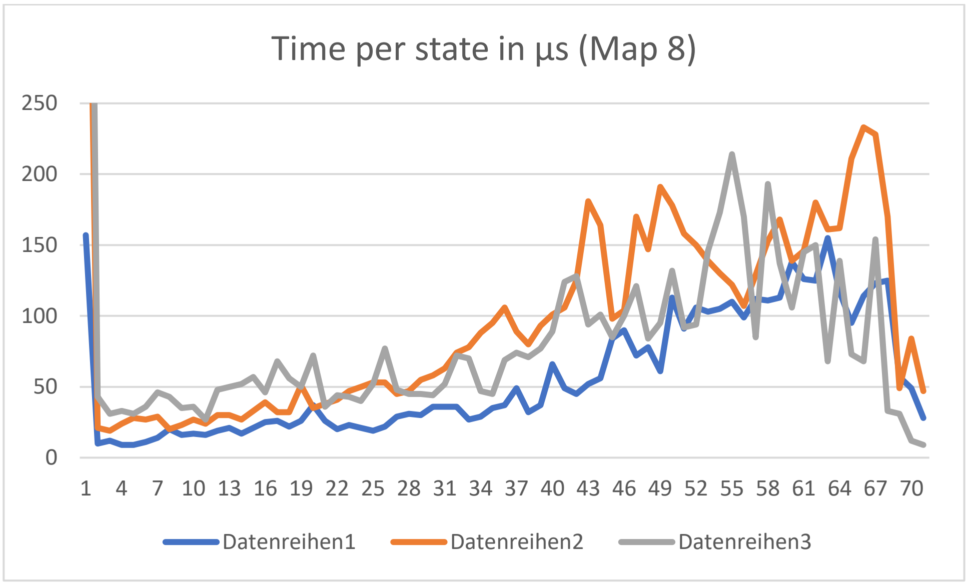

States (Map 30)
3.000
2.500
2.000
1.500
1.000
500
0
1
13
25
37
49
61
73
85
97
109
121
133
145
157
169
181
193
205
217
229
241
253
265
277
289
301
Datenreihen1
Datenreihen2
Datenreihen3

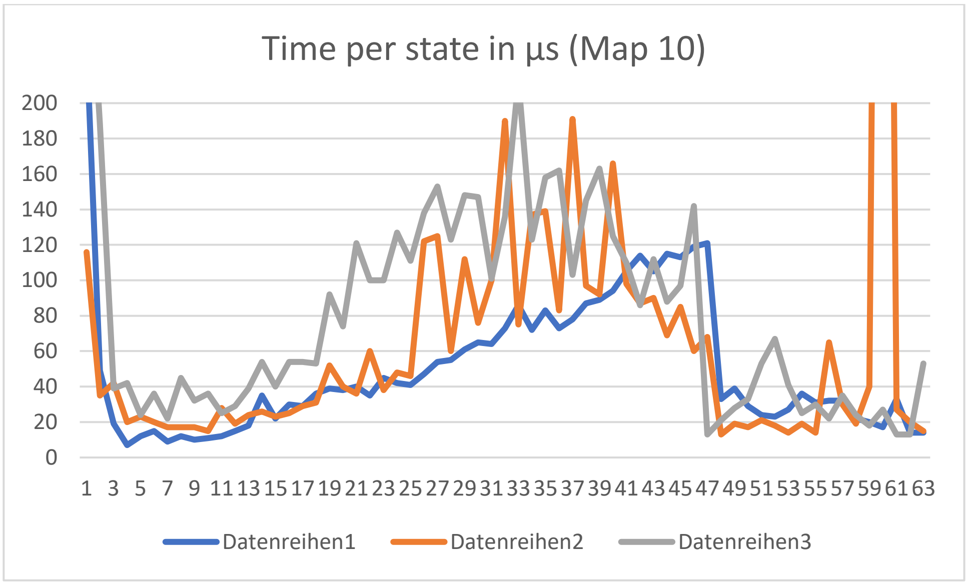
Times per state in µs for games with a 1s time limit.
Datenreihe1: no sort (blue)
Datenreihe2: with sort (orange)
Detenreihe3: with sort and beam of 5 (gray)

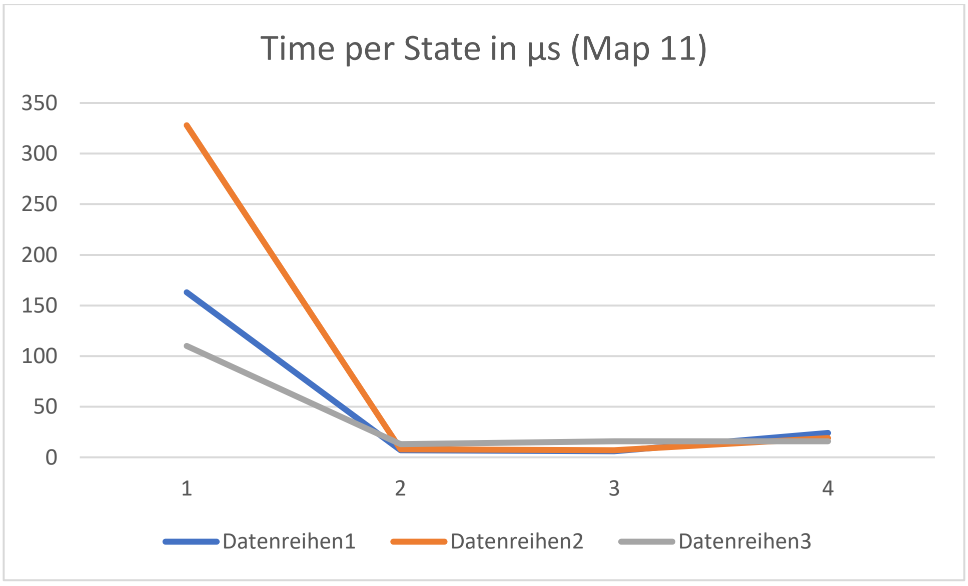


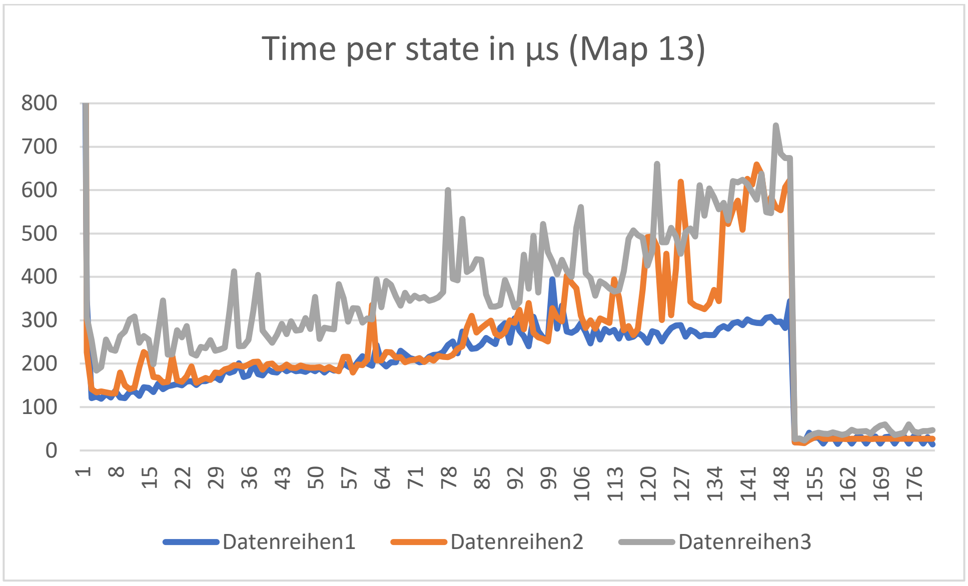


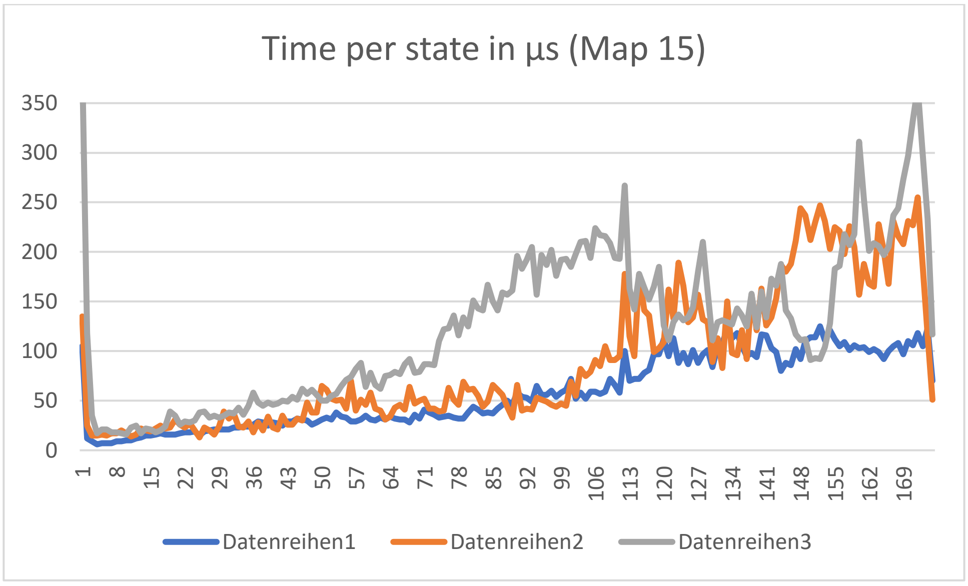


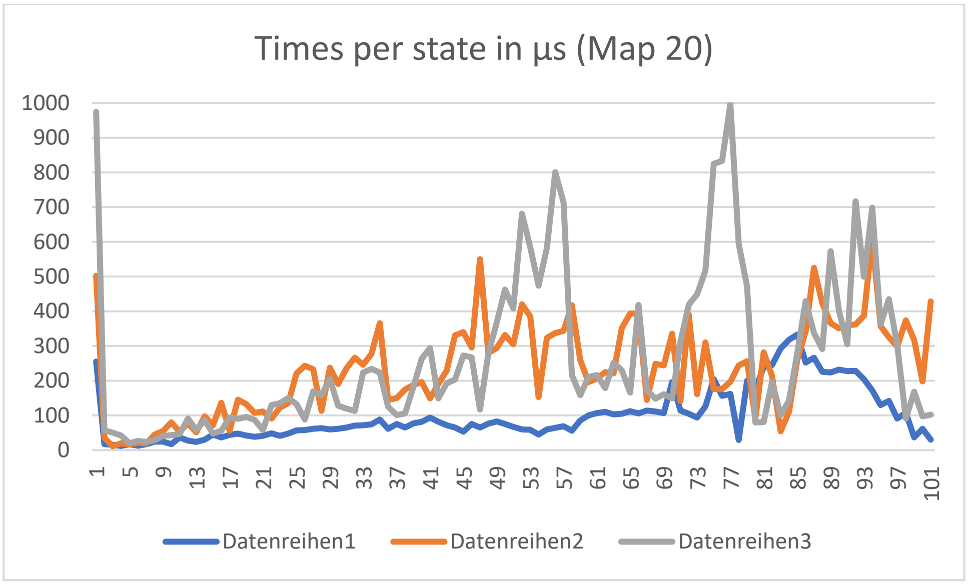

Time per state in µs (Map 8)
250
200
150
100
50
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70
Datenreihen1
Datenreihen2
Datenreihen3

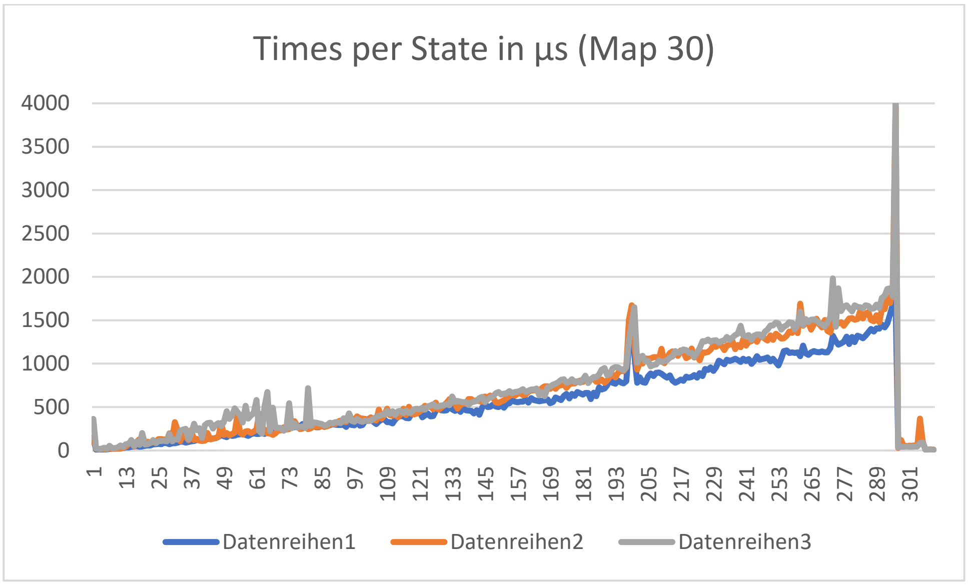

Time per state in µs (Map 10)
200
180
160
140
120
100
80
60
40
20
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63
Datenreihen1
Datenreihen2
Datenreihen3


Time per State in µs (Map 11)
350
300
250
200
150
100
50
0
1
2
3
4
Datenreihen1
Datenreihen2
Datenreihen3

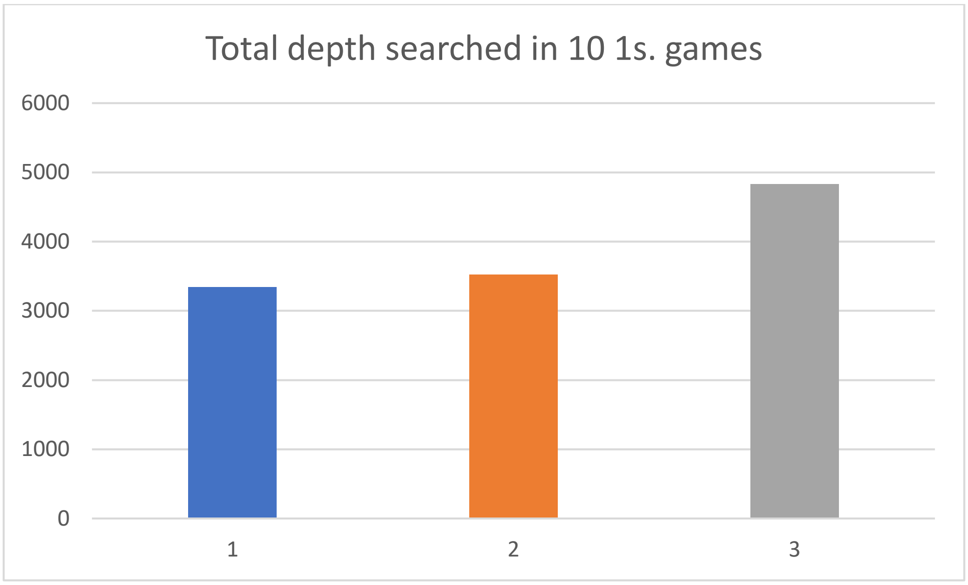

Time per state in µs (Map 13)
800
700
600
500
400
300
200
100
0
1
8
15
22
29
36
43
50
57
64
71
78
85
92
99
106
113
120
127
134
141
148
155
162
169
176
Datenreihen1
Datenreihen2
Datenreihen3


Time per state in µs (Map 15)
350
300
250
200
150
100
50
0
1
8
15
22
29
36
43
50
57
64
71
78
85
92
99
106
113
120
127
134
141
148
155
162
169
Datenreihen1
Datenreihen2
Datenreihen3


Times per state in µs (Map 20)
1000
900
800
700
600
500
400
300
200
100
0
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57
61
65
69
73
77
81
85
89
93
97
101
Datenreihen1
Datenreihen2
Datenreihen3



Times per State in µs (Map 30)
4000
3500
3000
2500
2000
1500
1000
500
0
1
13
25
37
49
61
73
85
97
109
121
133
145
157
169
181
193
205
217
229
241
253
265
277
289
301
Datenreihen1
Datenreihen2
Datenreihen3



Total depth searched in 10 1s. games
6000
5000
4000
3000
2000
1000
0
1
2
3