Softwarepraktikum SS 2016

Assignment 4

Group 3

Eric Wagner	344137	eric.michel.wagner@rwth-aachen.de
Stefan Rehbold	344768	Stefan.Rehbold@rwth-aachen.de
Sonja Skiba	331588	sonja.skiba@rwth-aachen.de

Task 1
Implementation of move sorting

The first task of this assignment was to implement a move sorting in combination with alpha beta pruning, while making sure we can turn it off at a later stage. As this task only really makes sense after the implementation of iterative deepening, we implemented them also in that order.

To be able to turn move sorting off we did save our old alpha beta algorithm and execute that in case we do not want to use move sorting. We decided that the algorithm are complicated enough and that we did not need more if-clauses, especially because the performance is also important. To switch off move sorting you can use the command line argument -a or –algorithm (-h for an explanation).

To implement the move sorting algorithm we use 2 lists. One for the current iteration and one for the iteration with an increased search depth. When we search with an depth of 1, thus executing the first iteration of iterative deepening the fill the list for a first time. For every move we check if it is better than the best move so far. If that is the case we put the element in the front of the list. Otherwise it will be placed at the end of the list. The only thing we can be sure about, is that the best move occupies the first place in the list. So we only get a pseudo-sorted list, but avoid the time to sort the list.
In every following iteration, in the root we iterate through the list, from front to back, and generate the list for the next iteration with the same method we used before.

What we still want to test, but has no time for this week is how we will perform with a completely sorted list and what will happen if we sort the moves for depth 2 and 3 as well.

Task 2
Benchmarking move sorting

The following tests where executed on an ubuntu distrubution run in an VM with 2 Gb of DDR4 RAM and an i7-6700HQ skylake processor.

In the following analyse we do concentrate on the amount of nodes generated by the search tree and the time it took the compute a move. The other benchmarks we implement were, as expected, not affected by move sorting in a notable way.

Except the first map, every map is analysed in 3 different states. The first one being the initial state, the second one in the middle of a game (at least 30% of stones placed). The last possible state is and end game state where 90% of the cells are occupied.
The maps taken for the benchmark partially taken from the server logs and partially created by us.
The results are discussed at the end and not after every map as the results should be similar for the most parts.

A list of the maps used for the tests:

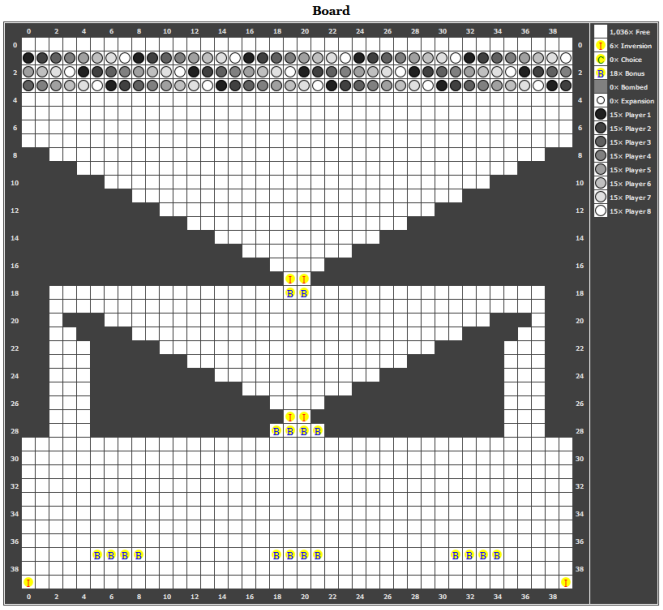


Figure 1: FunnelOfDeath

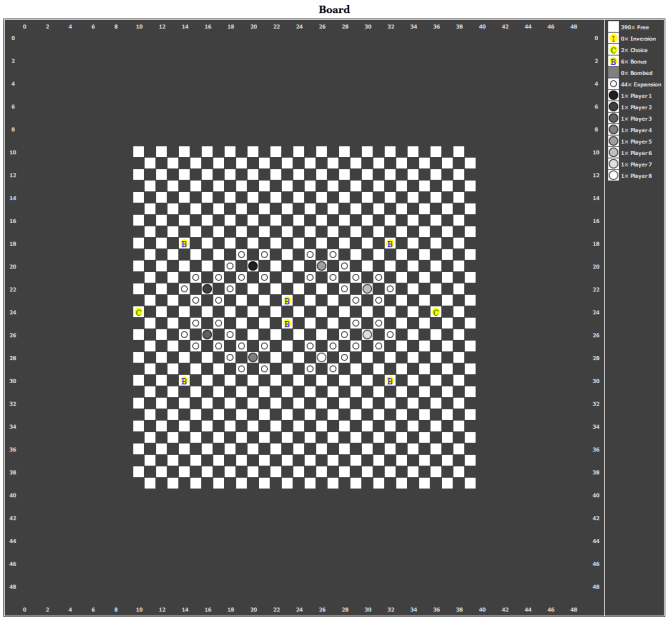


Figure 2: g3_2

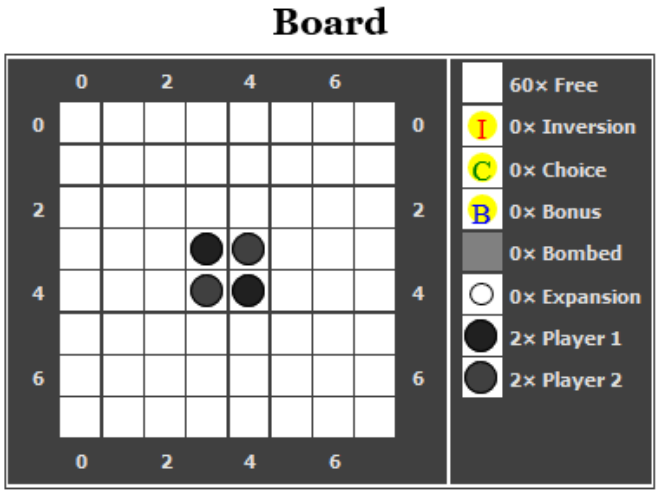


Figure 3: Standart

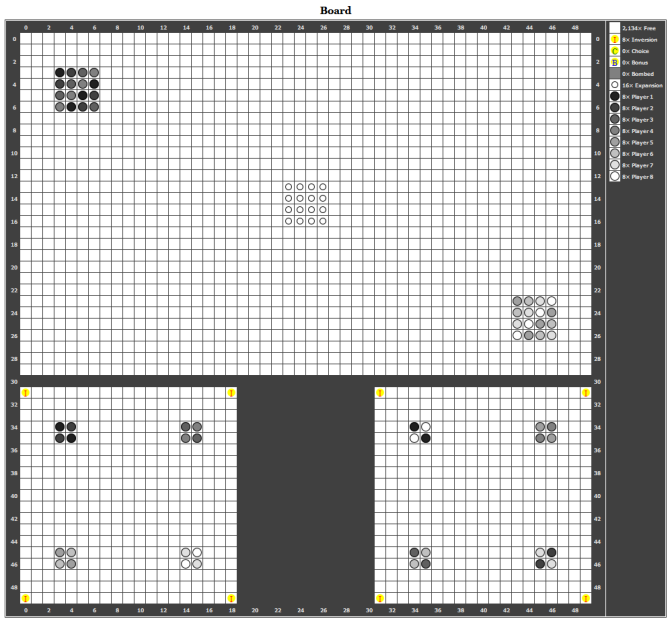


Figure 4: TripleOfTraps

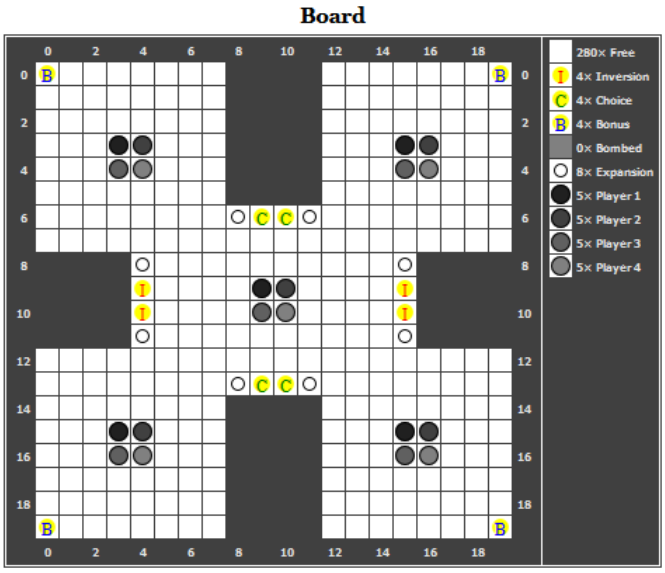
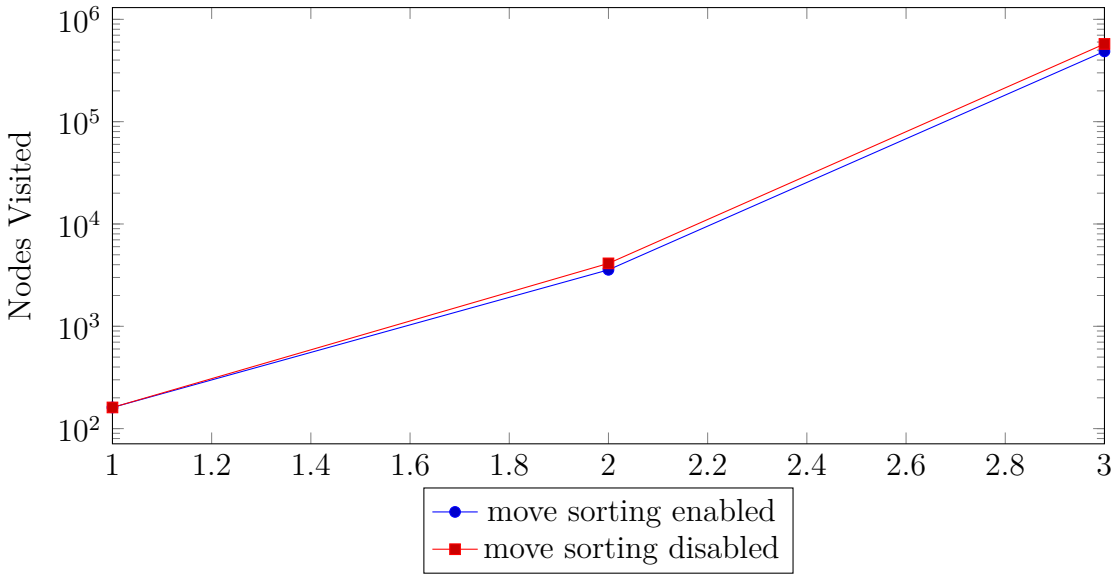

Board
280× Free
4× Inversion
4× Choice
4× Bonus
0× Bombed
8× Expansion
5× Player 1
5× Player 2
5× Player 3
5× Player 4

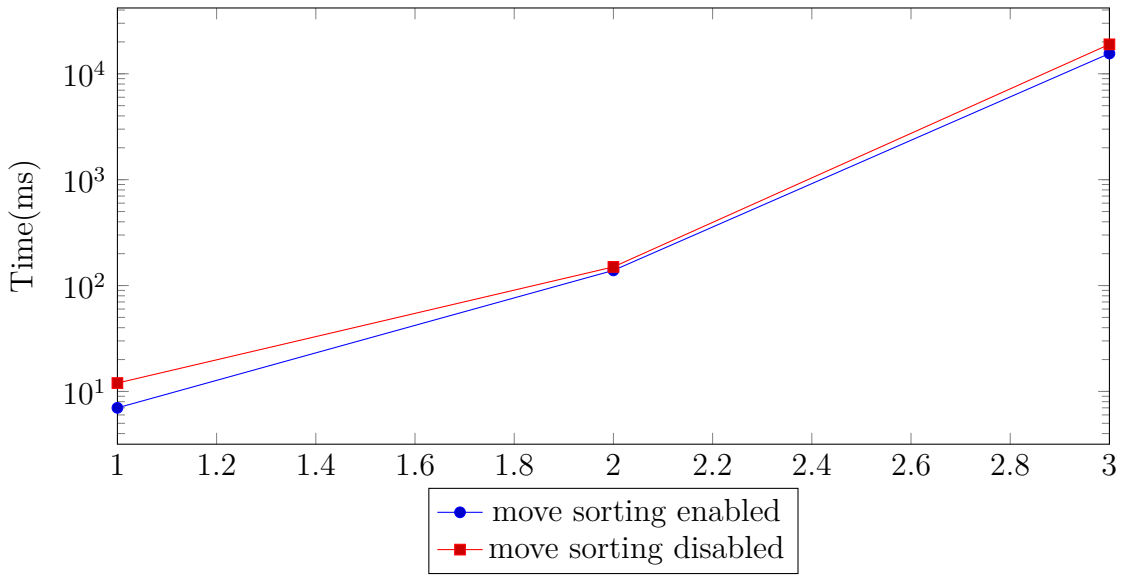
Figure 5: g5_1

Map: Funnel of Death

State: Start

Results:

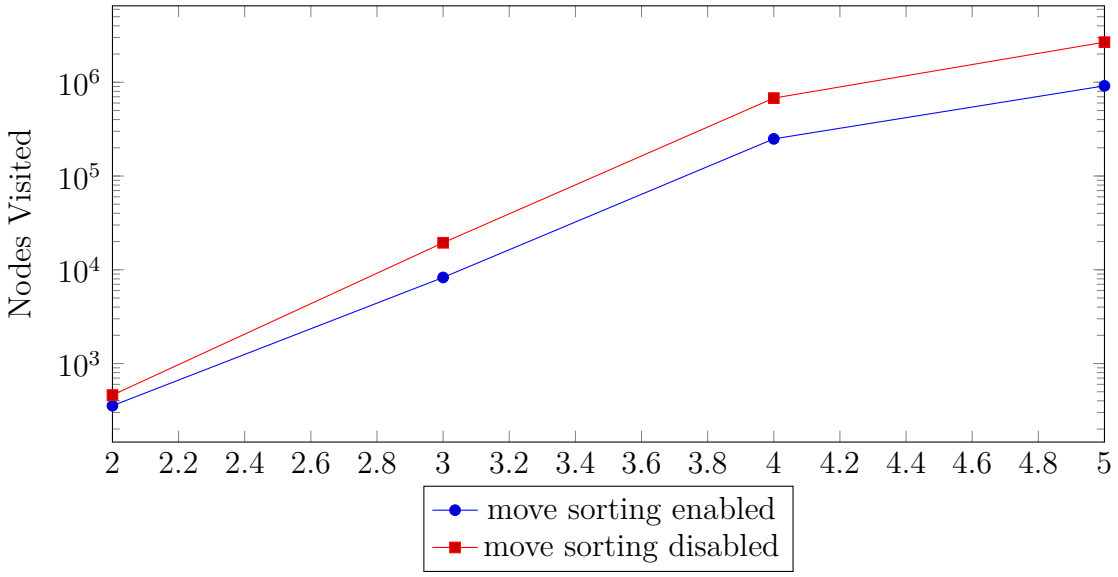


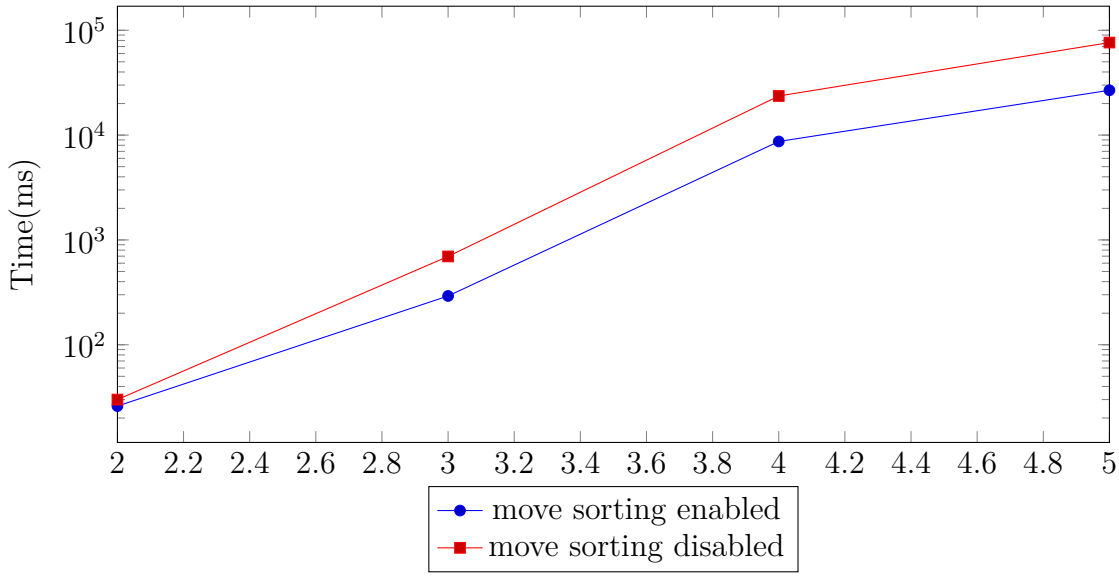


Map: g3_2

State: Start

Results:

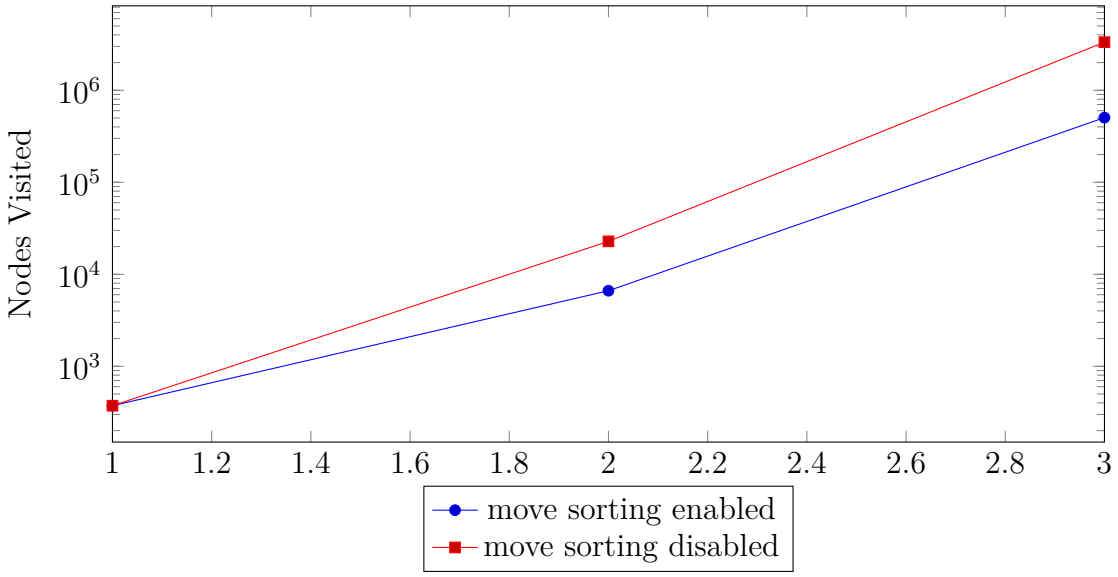


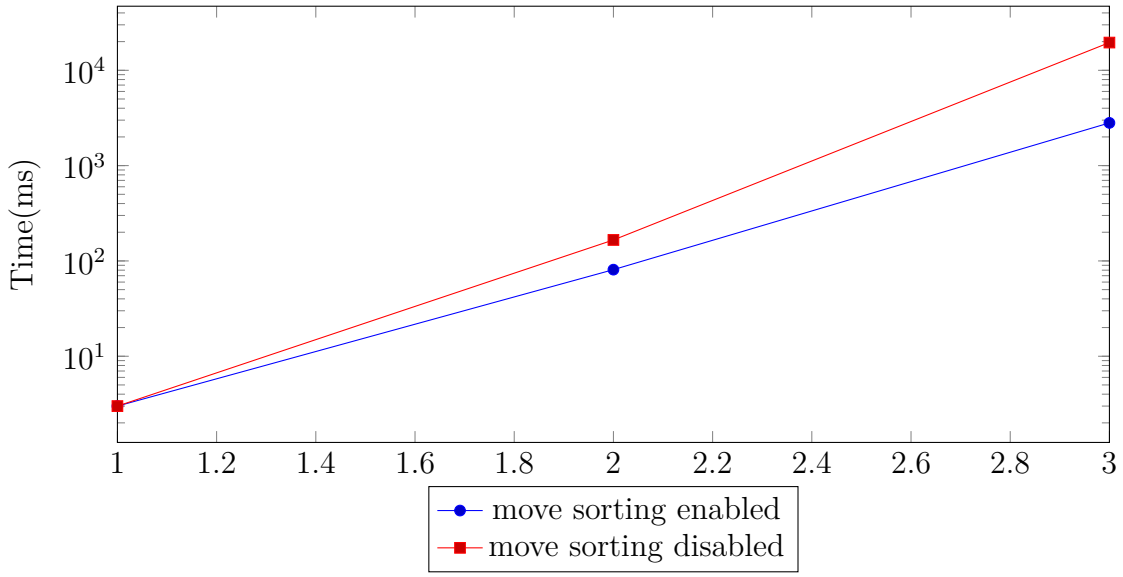


Map: g3_2

State: End

Results:

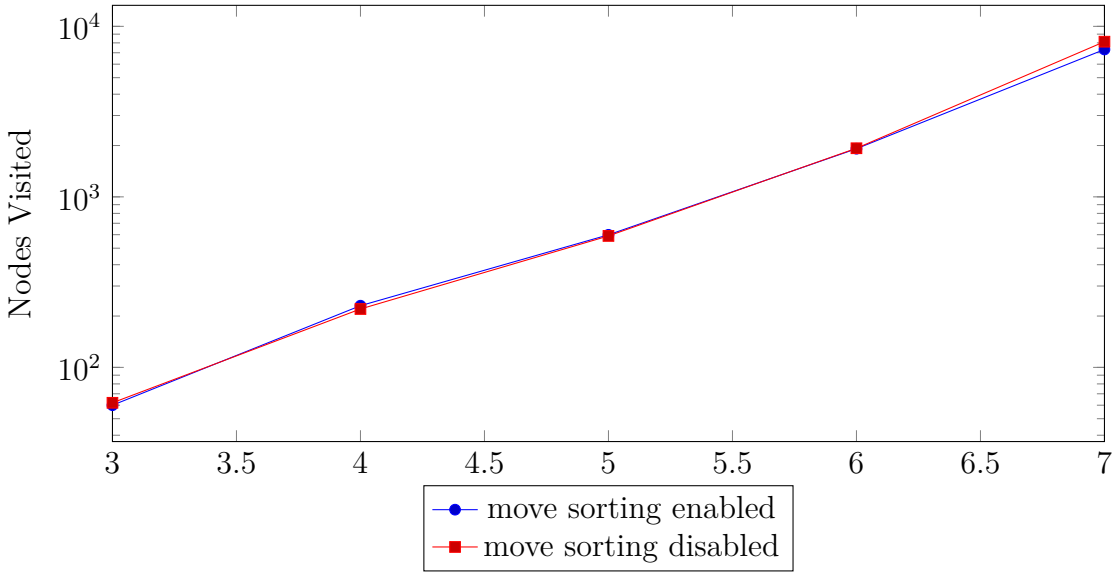


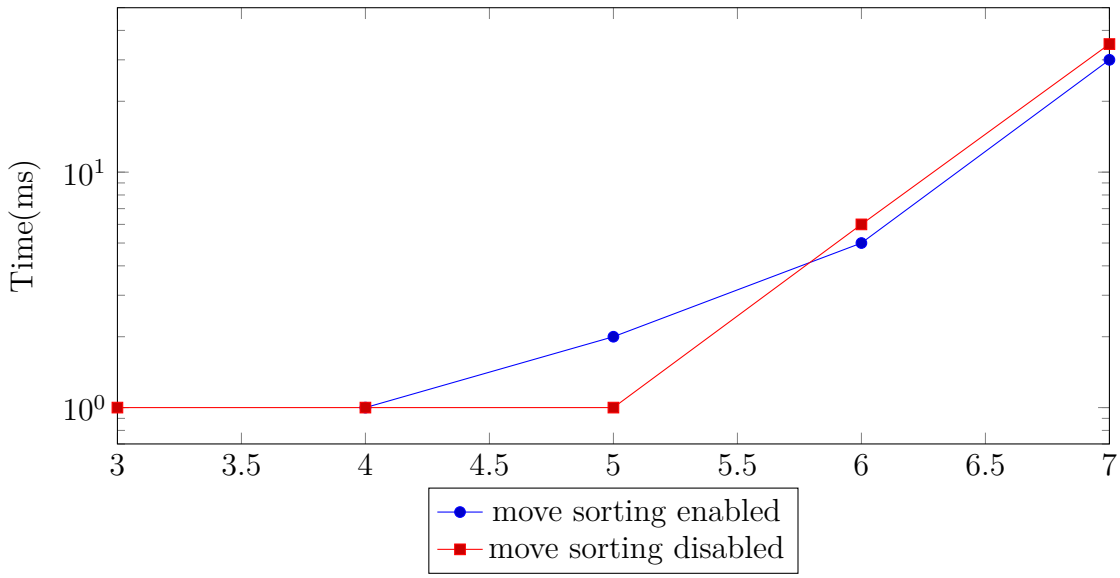


Map: Standard

State: Start

Results:

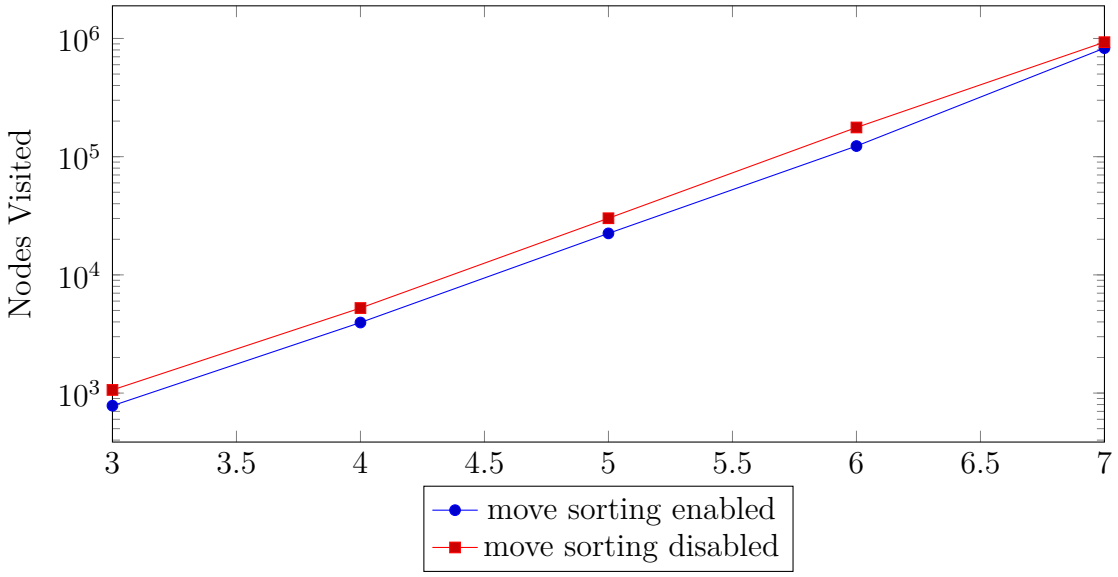


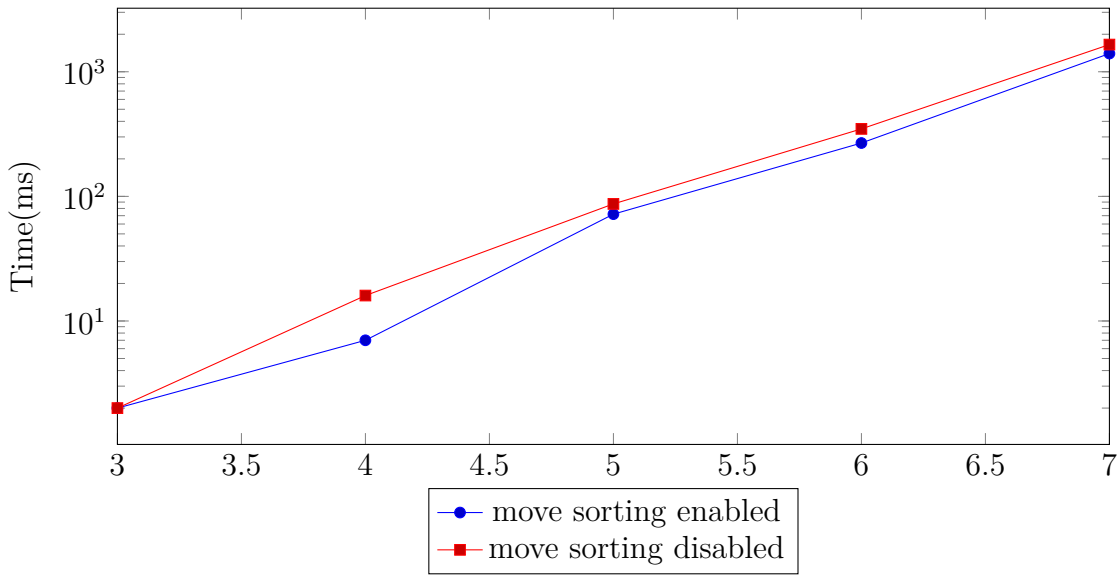


Map: Standard

State: Mid

Results:

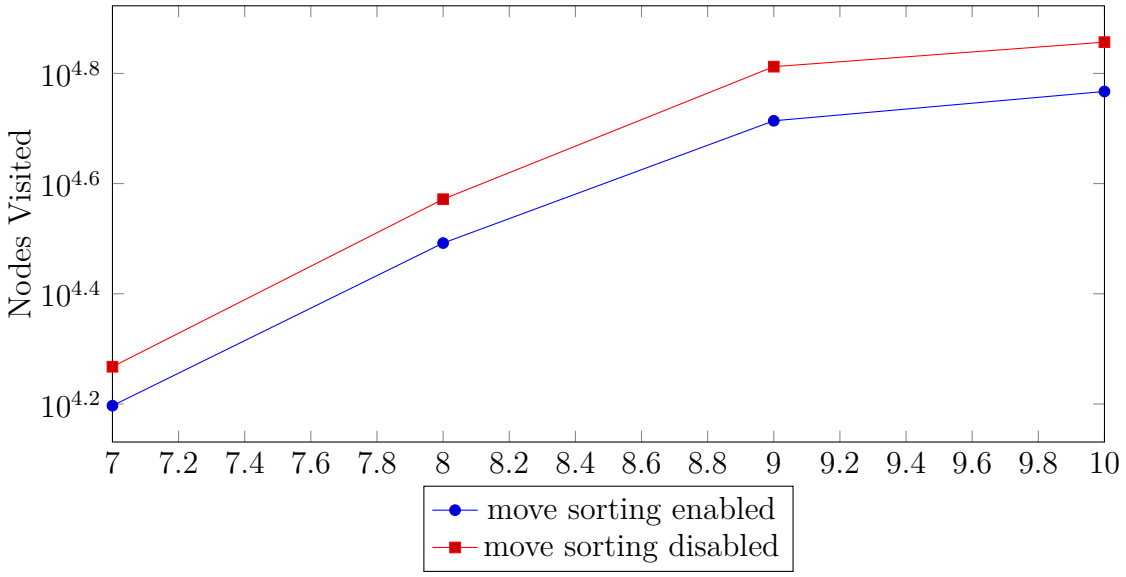


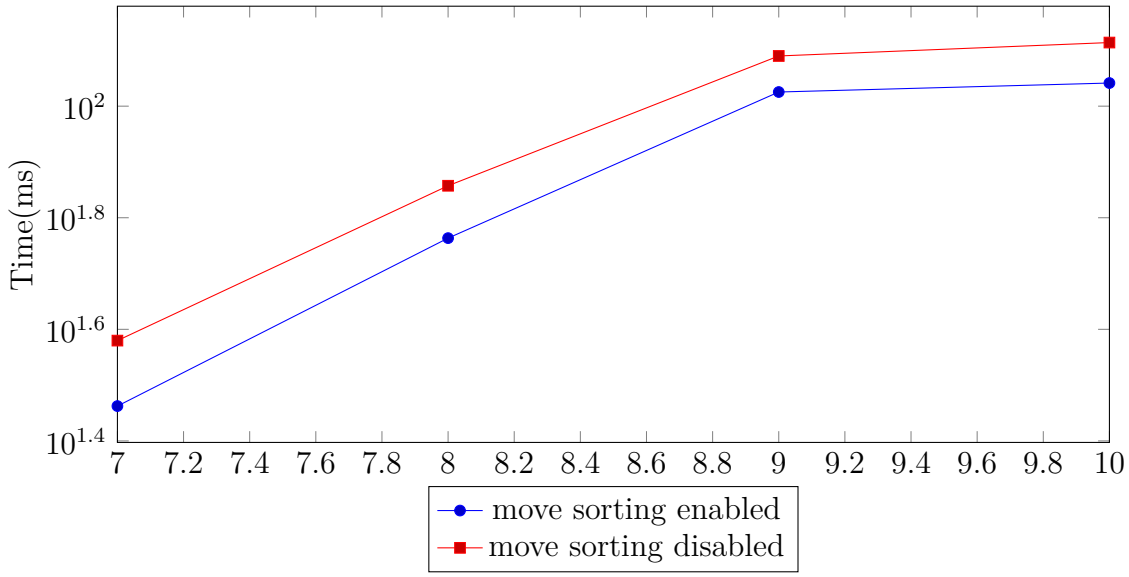


Map: Standard

State: End

Results:

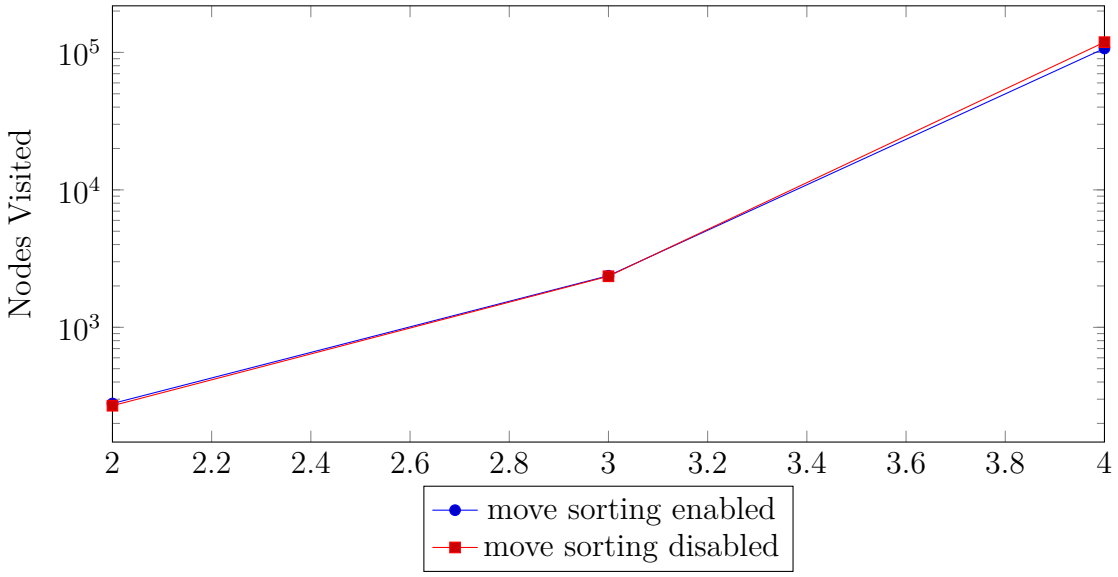


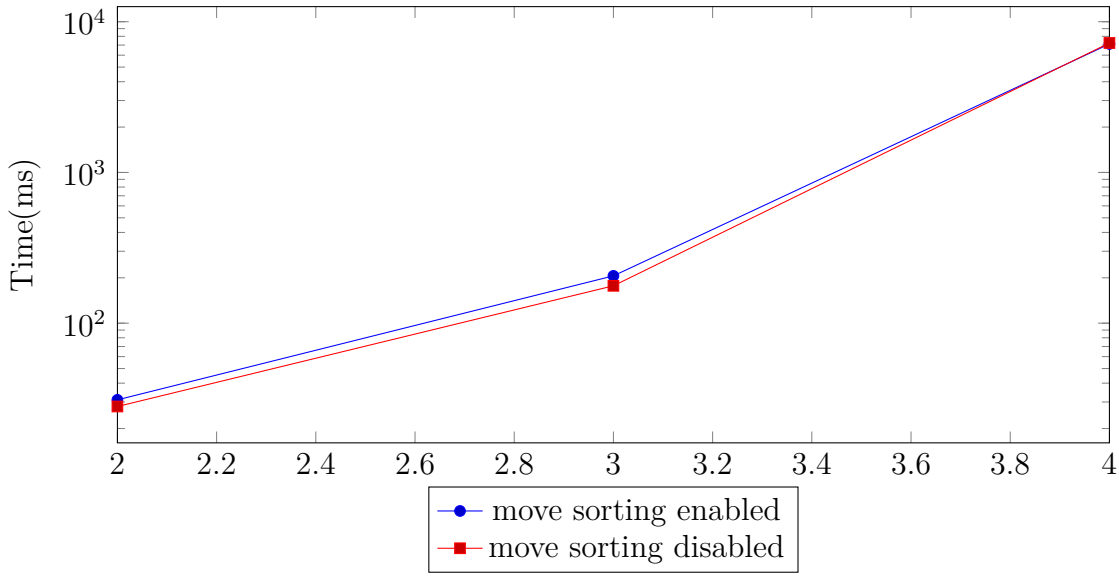


Map: TripleOfTraps

State: Start

Results:

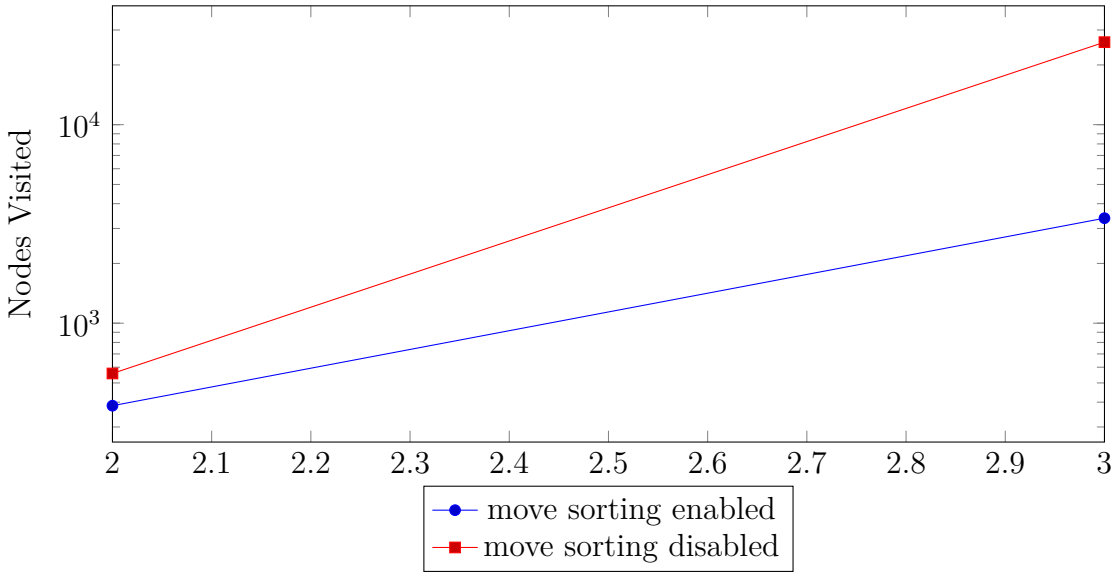


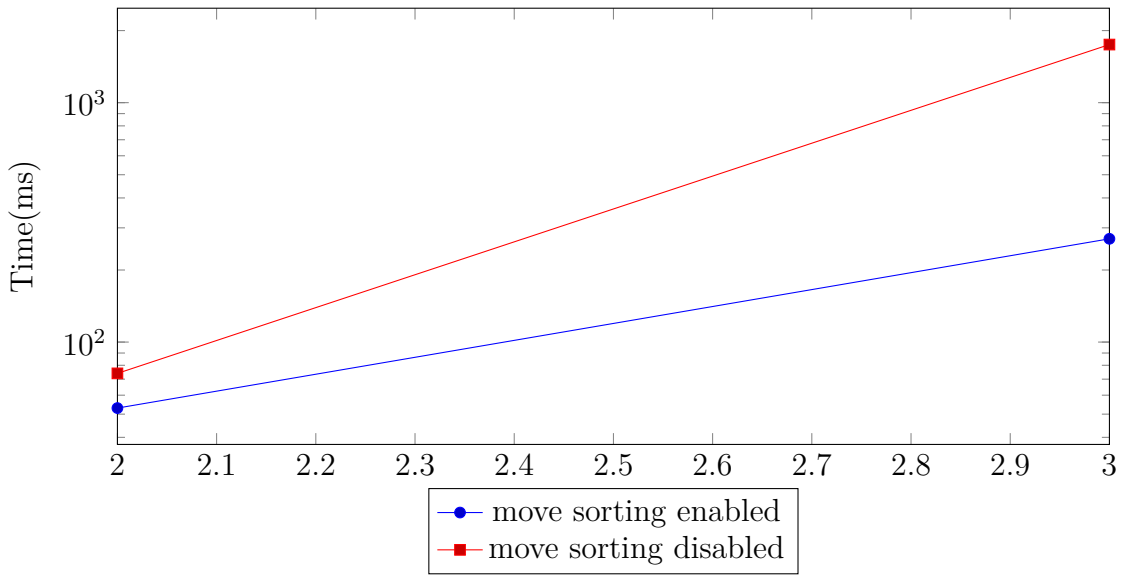


Map: TripleOfTraps

State: Mid

Results:

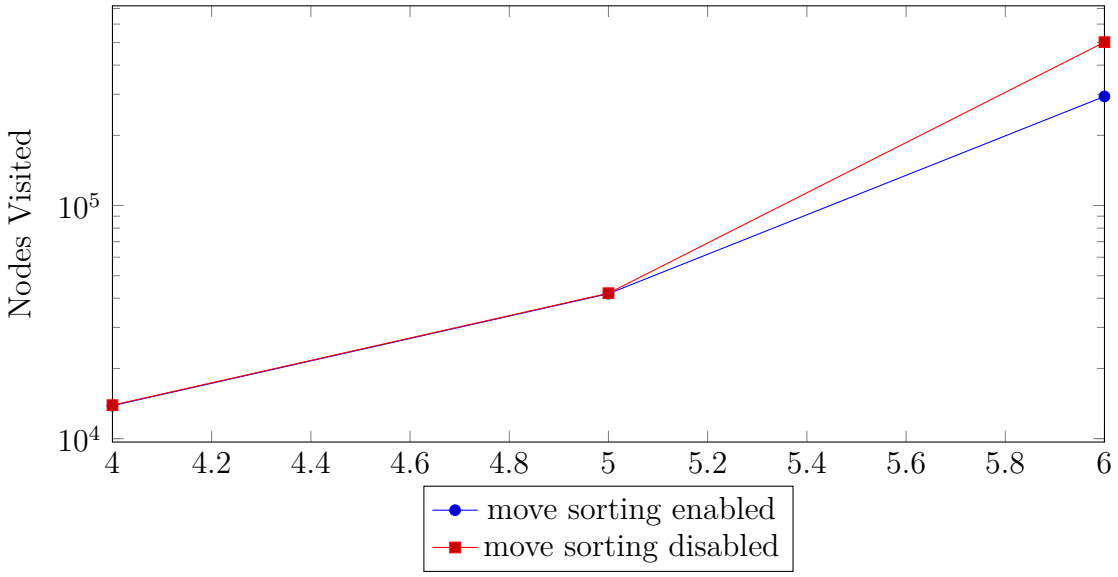


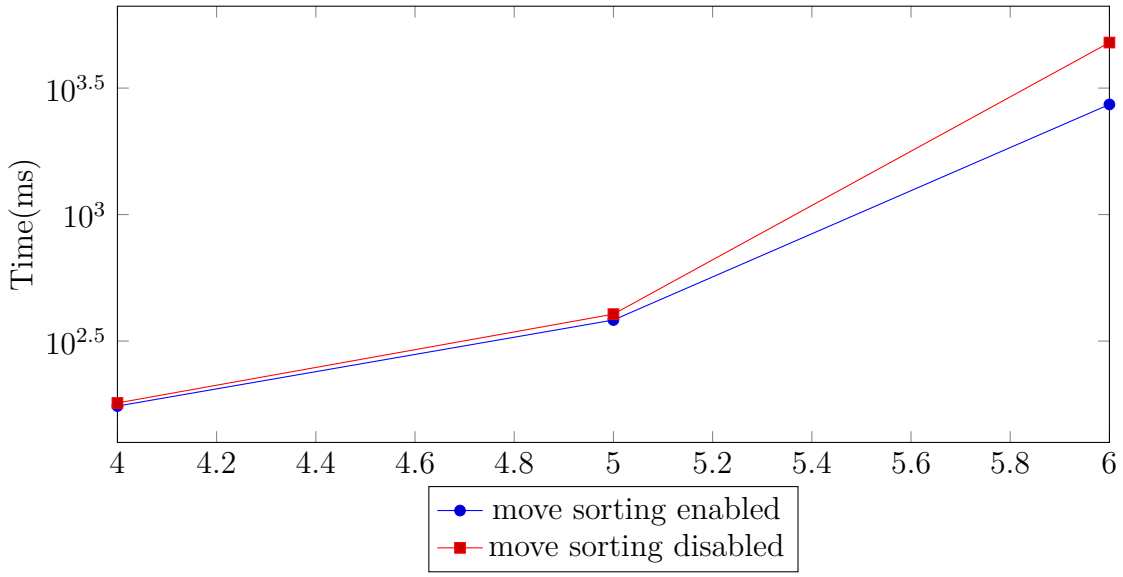


Map: g5_1

State: Start

Results:

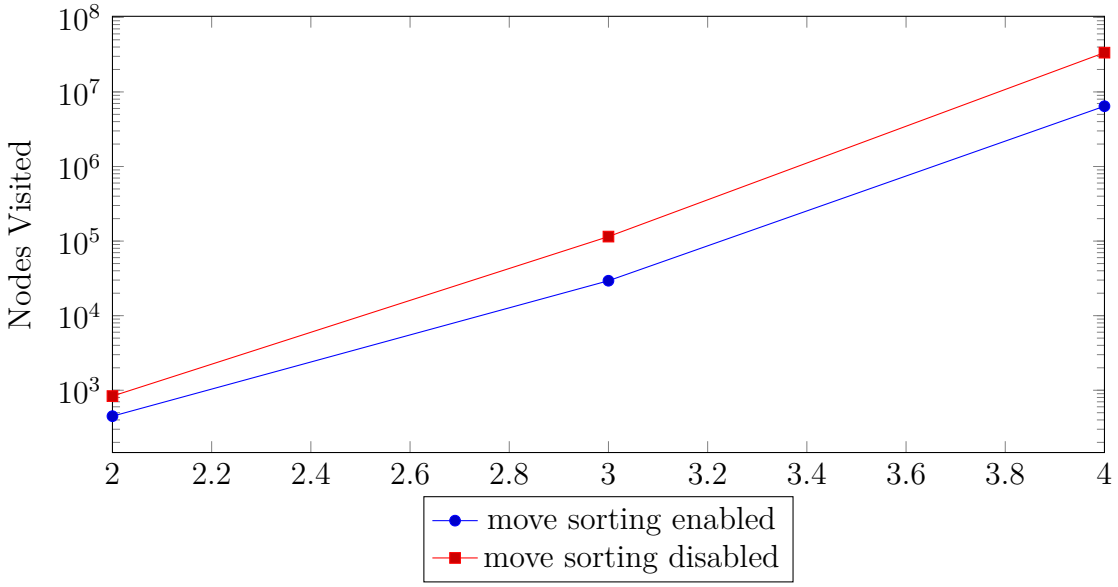


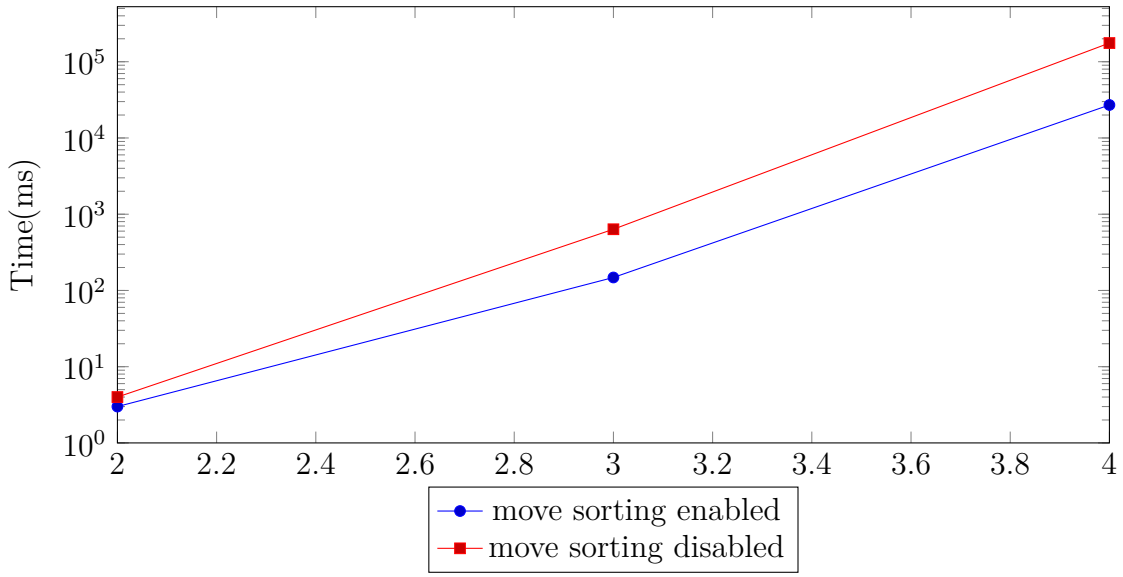


Map: g5_1

State: End

Results:





Analyse

Enabling move sorting is for almost every case more efficient. Overall move sorting seems to be nearly equal to normal alpha beta pruning at low depth and faster at higher depth. The difference is noticeable but not extraordinary.
Exceptions are:

- At depth 1 move sorting is slower. This has to do with the overhead of managing the lists.
- States at which nearly all cells are occupied *(Standard(end state))*. Here alpha beta pruning without move sorting is faster. This lets use suppose that move sorting is only useful if the game tree is in fact growing exponentially.
- *(TripleOfTraps(mid state))* In this case move sorting is over an order of magnitude faster. This might have to with an lucky order of moves.
- *(Standart(start state))* During this evaluation we found that move sorting is fast for depth 6 and 7. This seems strange but can have to do with us only using a pseudo sorted list and an unlucky order of moves.

Task 3:
Implementation of Iterative Deepening

The goal of this task was to implement iterative deepening and use it in a second phase to adjust our program to be able to handle time limits.
For this we will execute the move search multiple, while each time increasing the search depth, starting at 1.
Due to the exponential growth of game trees, we do not loose too much time by regenerating the trees of smaller depth again and again.

For an average branching factor of b and depth d, we can calculate the average amount of nodes:

$$\sum_{k=1}^{d}(b^k) = \frac{b^{d+1}-b}{b-1}$$

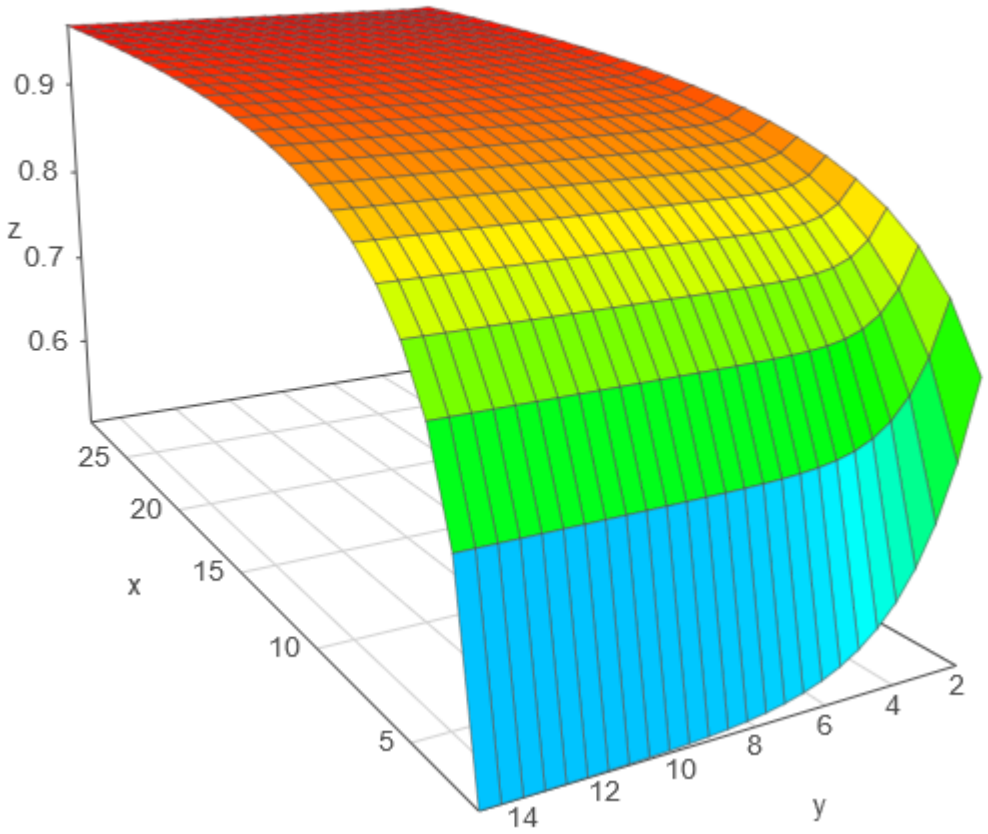
To execute the move search for depths 1 to $d-1$, we can determine approximate amount of nodes visited

$$\sum_{k=1}^{d-1}(\frac{b^{k+1}-b}{b-1}) = -\frac{b(-b^d+bd-d+1)}{(b-1)^2}$$

We can now compute the ratio between the nodes in the trees of depth 1 to $d-1$ and in the tree of depth d:

$$\frac{\frac{b^{d+1}-b}{b-1}}{\frac{b^{d+1}-b}{b-1} + (-\frac{b(-b^d+bd-d+1)}{(b-1)^2})}$$

In the following picture this ratio is visualized. We see that the tree of depth d take a large amount of the time, for bigger trees even more. So we only loose a small fraction of the time to compute the tree for depth 1 to d in advance.



x : stands for the branching factor b
y : stand for the search depth d
z : ratio
All computation were made with WolframAlpha
To plot the function we used the tool on the site of academo.org

With alpha beta pruning we would not generate all of these nodes, but the proportion would remain the same.

By using iterative deepening we have two advantages. The first one is that we can save what moves looked promising in a previous iteration and execute them first. Like this we expect to get the *alpha* and *beta* values closer together as they normally would and be able to prune away more of the tree. Effectively approaching the best-case runtime of alpha-beta pruning of $O(b^{d/2})$ (which is reached if we execute the best move first).
Further we have the advantage that after the first iteration has finished, we always

have move that we can return once the time has run out. So we avoid to return a move after only have analysed a fraction of the board.

This leads us to the second part of this task, the capability of reacting to a time limit. We achieved this by using the methode call

```
setitimer (ITIMER\_REAL, \&timer, 0).
```

The first argument is the type of the timer. Here we use a real-time timer as the server also count in real time. The second argument sets the time until the timer sends a signal. The last argument is not needed.

The timer arguments are set as following (in advance we have already subtracted 50 ms from the search time to account for time it takes to close all the recursion stacks, free the memory and send a message to the server):

```
timer.it\_value.tv\_sec = (searchTime/2)/1000;
timer.it\_value.tv\_usec = (searchTime/2)%1000)*1000;
timer.it\_value.tv\_sec = (searchTime/2)/1000;
timer.it\_value.tv\_usec = (searchTime/2)%1000)*1000;
```

The first 2 values define, in seconds and microseconds, when the timer should send a signal the first time. With the third and forth argument we set the interval at which the timer should keep firing after the first time. Before we leave our move search algorithm, we cancel the timer to make sure it does not go off at an unexpected moment and cause undefined behaviour. This is done by recalling the the setitimer methode with all the timer arguments zeroed.

The last step was to write the callback function to define what happens once the timer sends a signal. As the timer runs in a separate thread when executing the code of the callback function, we have to avoid race conditions. To achieve this we use flag of type **static volatile sig_atomic_t**.

The first call of the callback function will set the **hasHalfTimePassed**-flag, the second one the **hasTimePassed**-flag. The **hasHalfTimePassed**-flag is checked after every iteration. If we have less than half our time we can assume that we will not finish the next iteration. SO we will return immediately and save the time for the following turn. The **hasTimePassed**-flag is checked constantly during the move search to make sure we will catch it as soon as possible and have enough time to return the move found in the previous iteration.