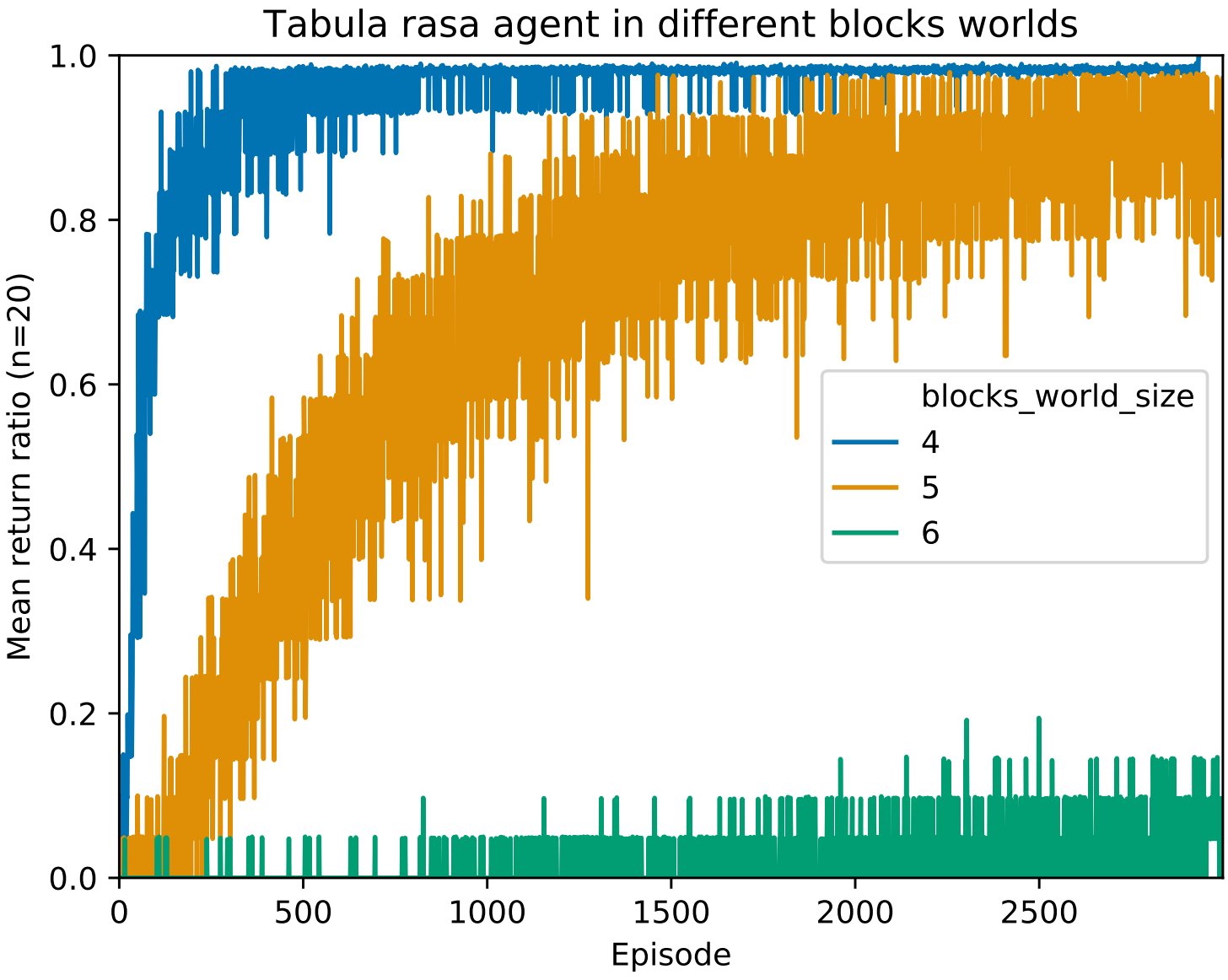
Experiment 1

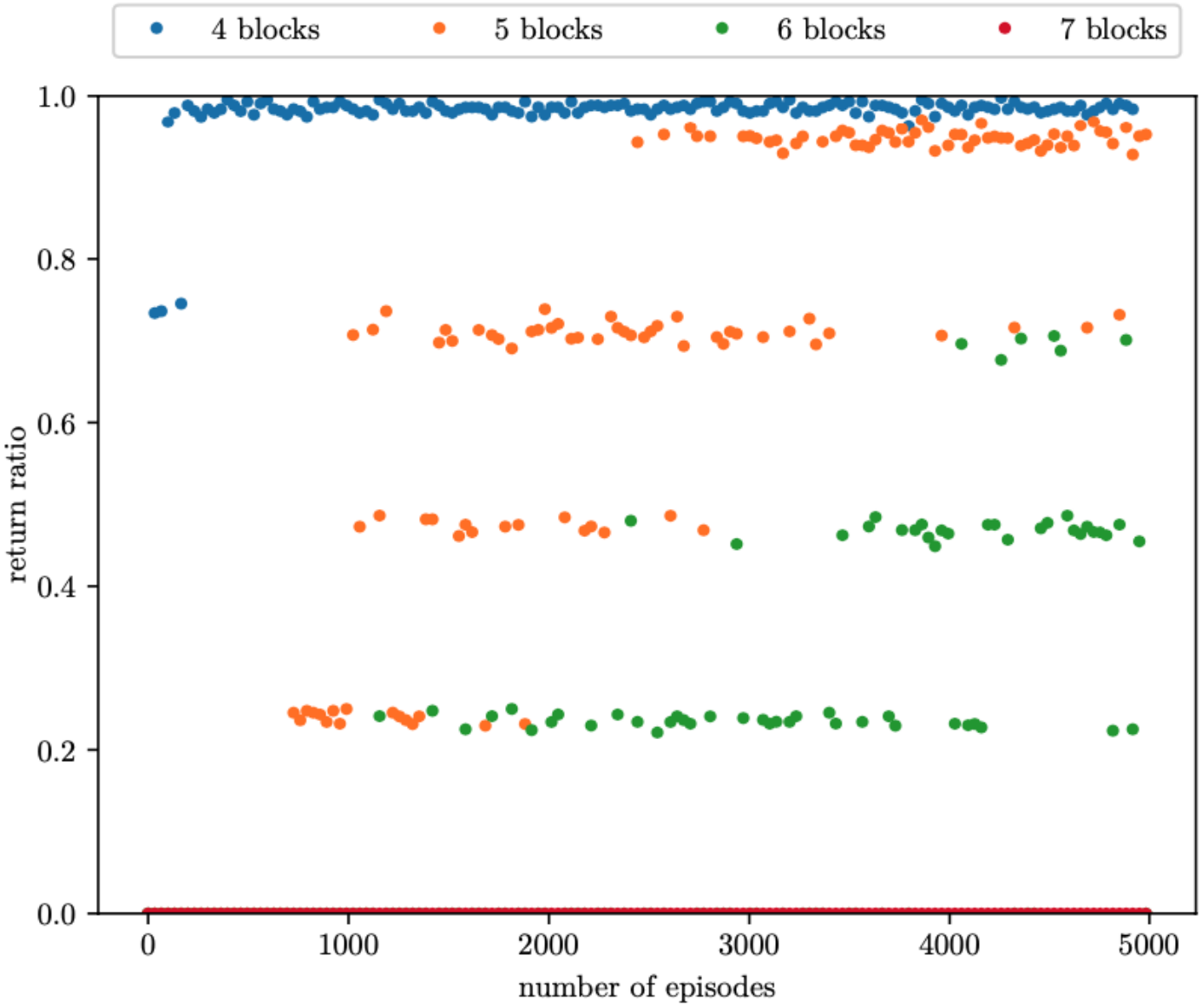
To reproduce the results from the bachelor's thesis

Experiment 1a



Experiment 1a

Comparison with bachelor's thesis

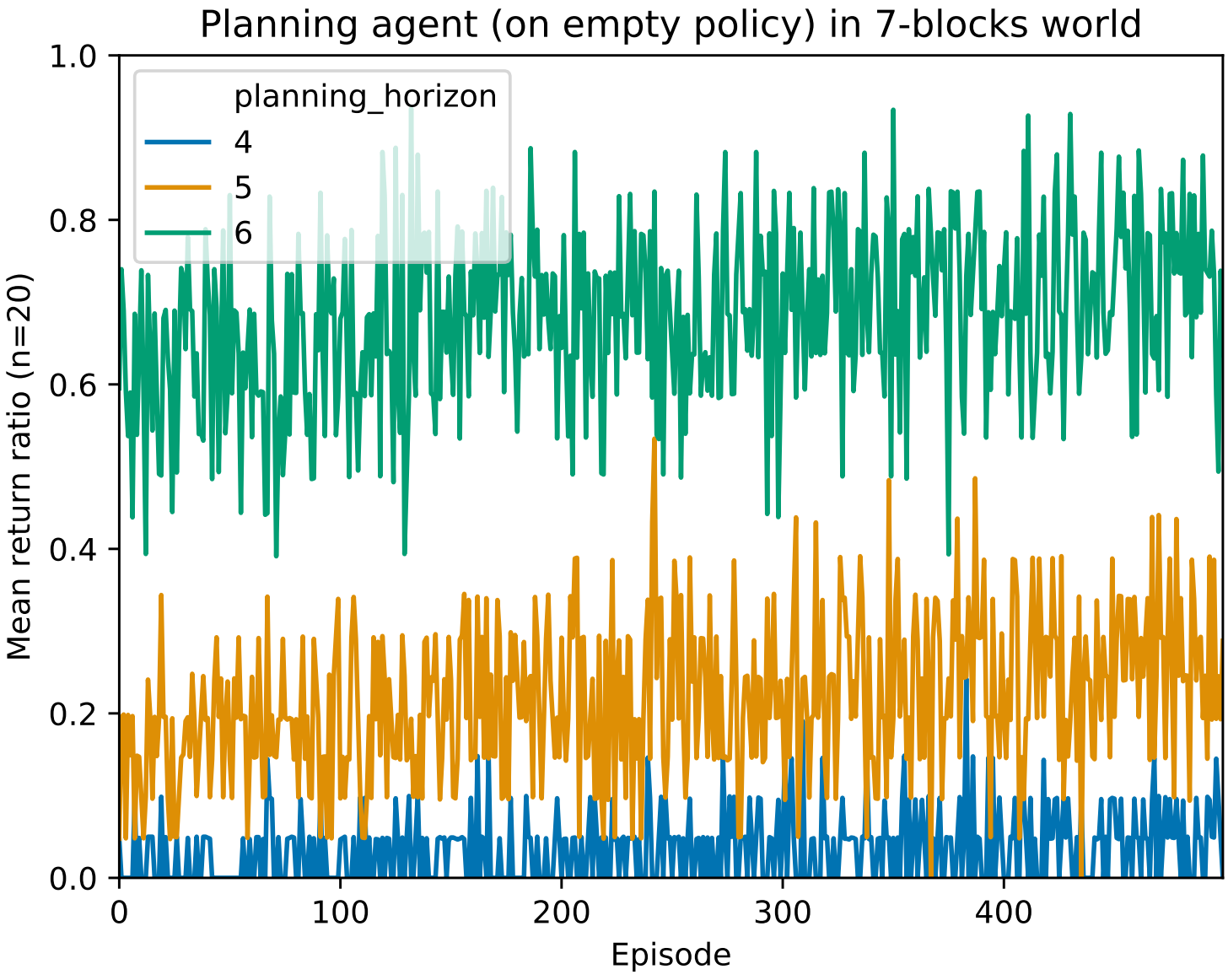


Experiment 1a

Remarks

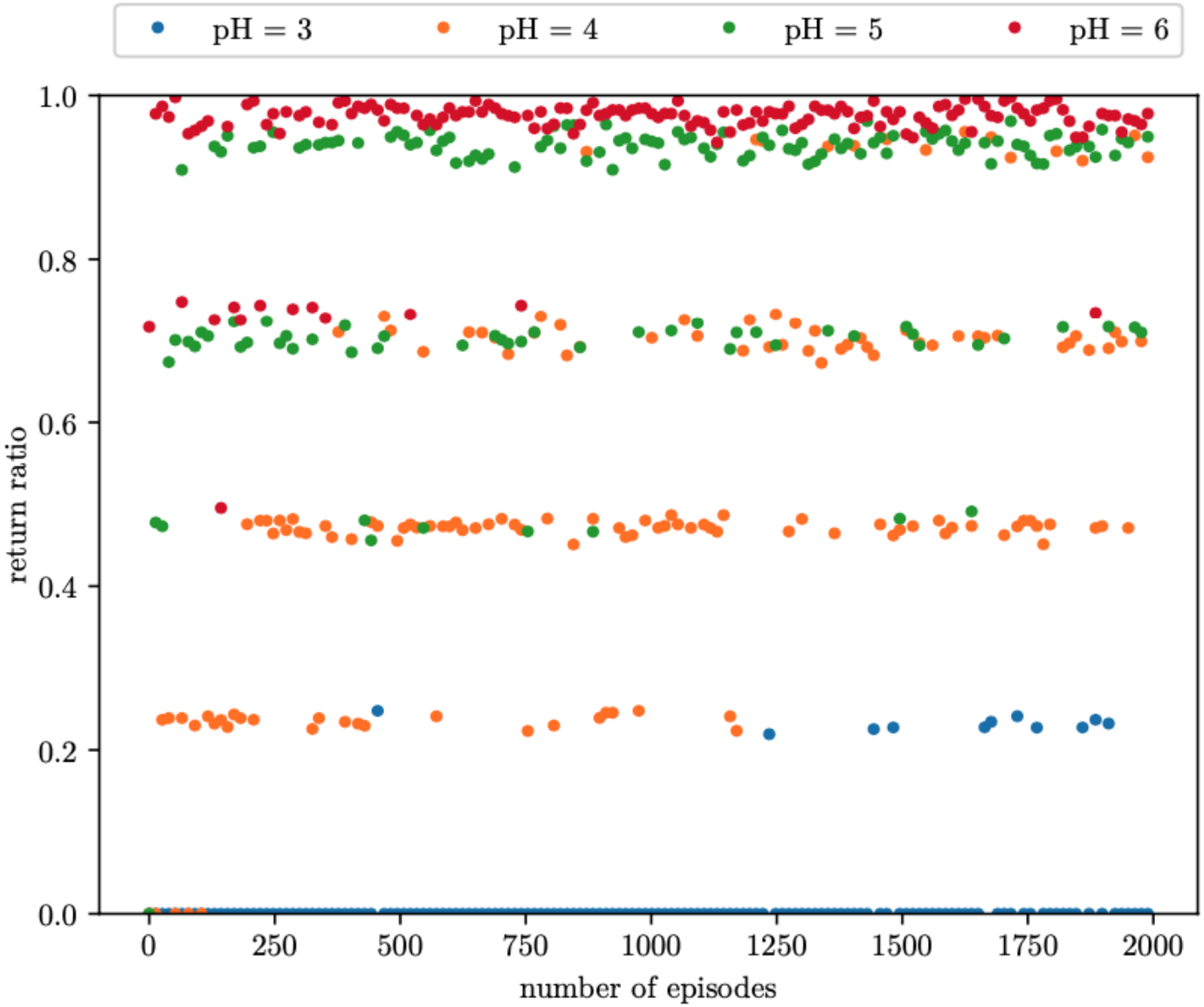
Results are in line with those of the bachelor's thesis.

Experiment 1b



Experiment 1b

Comparison with bachelor's thesis



Experiment 1b

Remarks

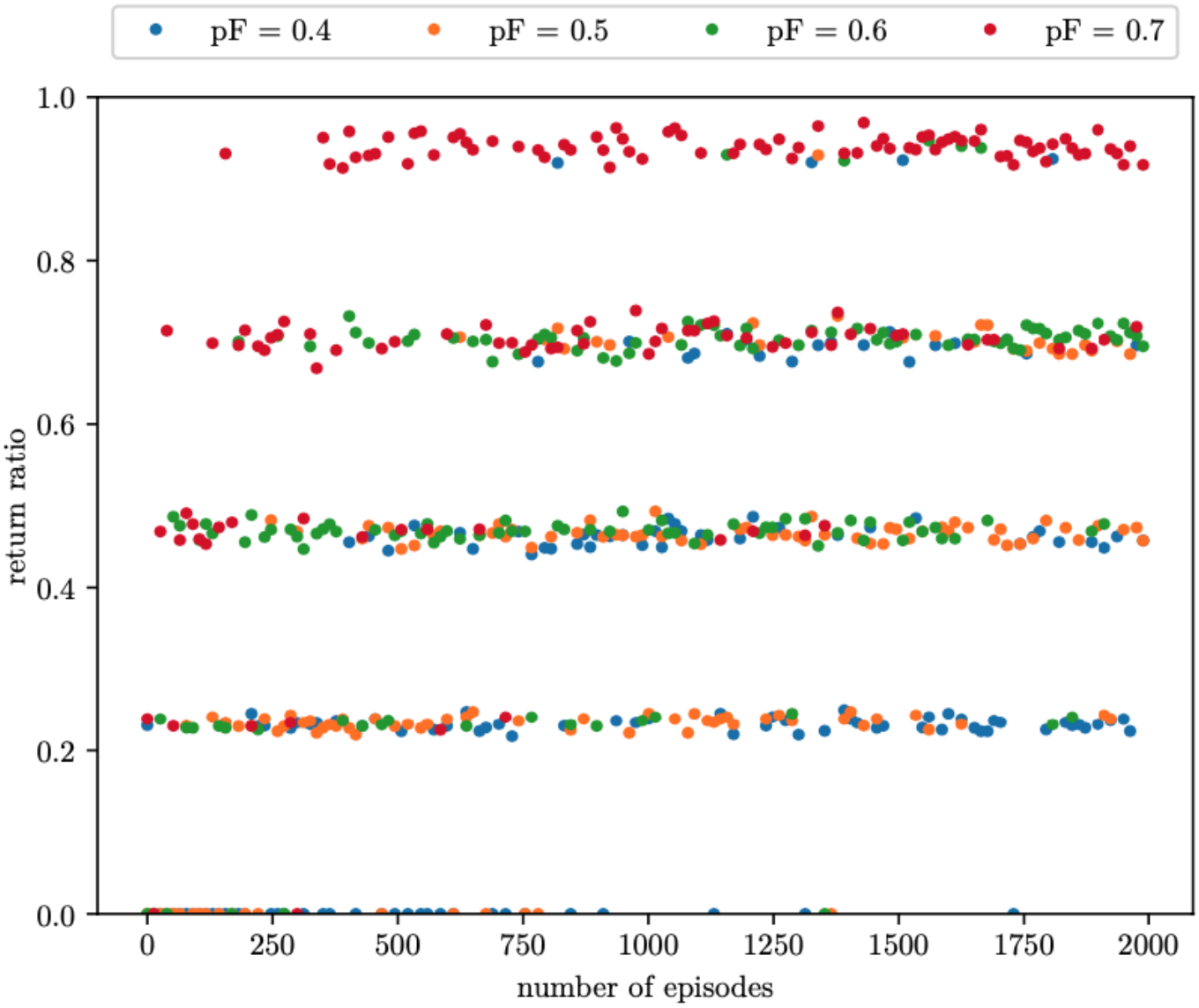
Compared to the bachelor's thesis, there seems to be very little progress. This is because here, only the first 500 (vs. 2000 in the thesis) are rendered. The results are probably similar to the thesis when the full 2000 episodes are rendered.

Experiment 1c

The original output (left) is too messy. Used running average to smooth the result and make it more readable (right).

Experiment 1c

Comparison with bachelor's thesis



Experiment 1c

Remarks

Only the first 1000 episodes were rendered (vs. 2000 in the bachelor's thesis). It's hard to say because of the noise, but the results look similar to those in the thesis.

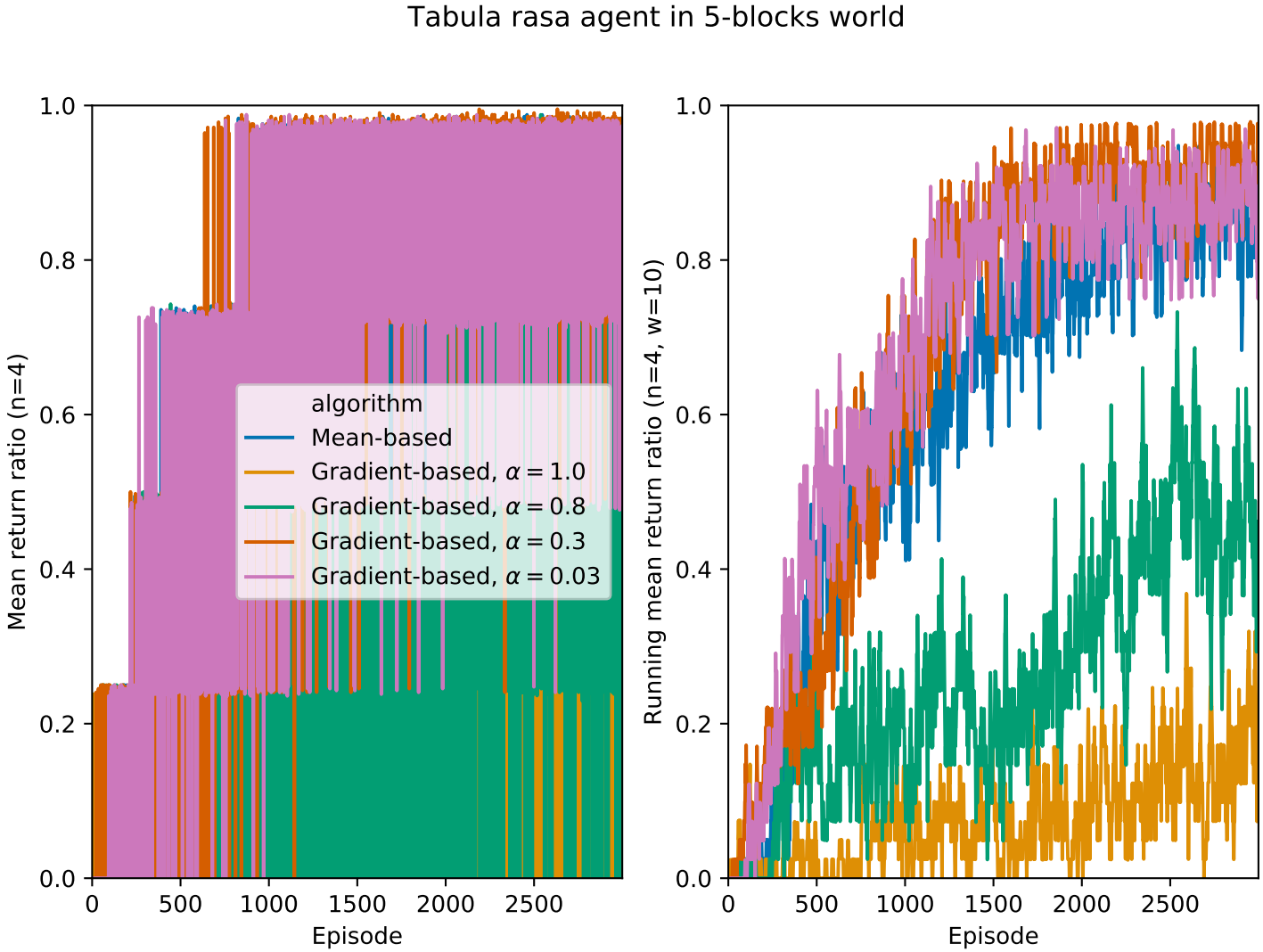
Conclusion

- Overall, the results seem to be in line with that of the bachelor's thesis.
- A low sample size ($n = 4$) may skew the results
- Generating results took ca. 4-13 hours. The blocks world size has a big impact on performance.
  - *Possible culprit:* Computing the return rate is expensive. It requires computing optimal plans for every step.
  - *Possible culprit:* Computing plans is expensive. For every state, the optimal return up to the planning horizon is computed, even if no answer set leads to a correct solution.

Experiment 2

To compare the mean-based First-visit Monte-Carlo method from the bachelor's thesis with a gradient-based every-visit monte-carlo method.

Experiment 2a



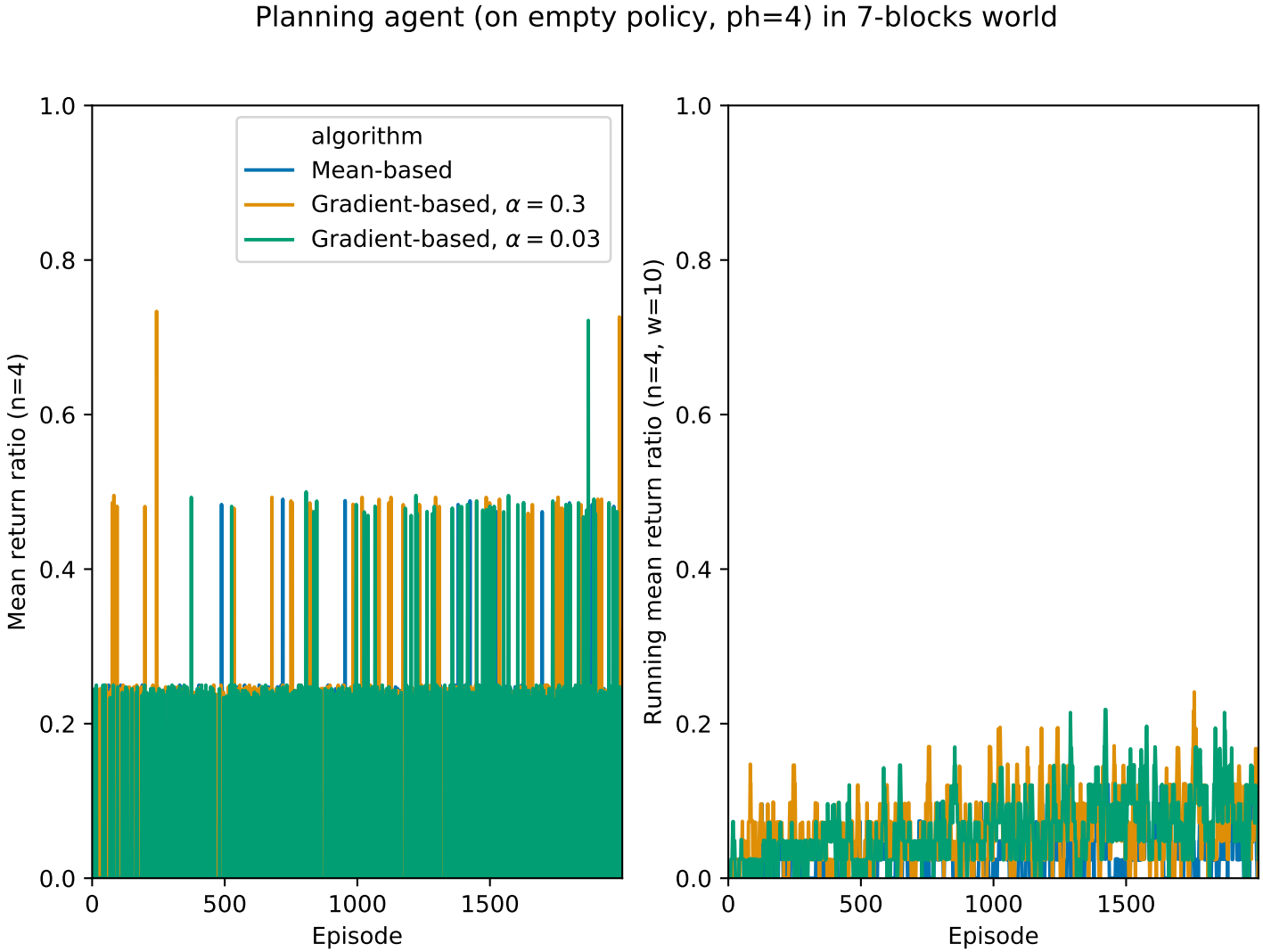
The original output (left) is too messy. Used running average to smooth the result and make it more readable (right).

Experiment 2a

Remarks

- The gradient-based algorithm at its best outperforms the mean-based algorithm.
  - This effect is caused by the difference between first-visit and every-visit Monte-Carlo methods, as can be shown when comparing a first-visit mean-based agent to a every-visit mean-based agent (which achieves similar performance as the gradient-based algorithm)
- There seems to be no significant difference in performance for $\alpha \in \{0.3, 0.03\}$
- However, there seems to be a decrease in performance for higher values ($\alpha \geq 0.8$)

Experiment 2b



The original output (left) is too messy. Used running average to smooth the result and make it more readable (right).

Experiment 2b

Remarks

- The low performance is disappointing, compared to the results of the bachelor's project (Experiment 1b, Comparison), where $pH = 4$ achieves positive results for two of the four runs.
  - With such a low sample size ($n = 4$), some randomness is expected. Maybe the result changes with larger sample sizes?
  - The low sample size is chosen because the experiment was expensive to compute ($\sim$ 13 hours for $n = 4$).
- The mean-based agent is equally bad. However, the mean-based agent worked fine in Experiment 1. This suggests that the result is not due to a programming error.
- Probably a good idea to repeat the experiment with a bigger planning horizon and more samples.

Conclusions

- Overall, the gradient-based method seems to work well.
- In combination with planning, the results are ambigous and should be refined.
- Low sample sizes are a problem, but it takes very long to run trials with larger sample sizes.

Experiment 3

The previous experiments took very long to complete. This experiment is for profiling the framework and identifying possible bottlenecks.

Experiment 3a

Tabula-rasa agent in a 7-blocks world, 150 episodes

Tue Aug 18 17:46:26 2020 exp3a_profile_raw.txt

```
         222241983 function calls (222234281 primitive calls) in 193.932 seconds

   Ordered by: internal time
   List reduced from 2716 to 20 due to restriction <20>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 19786382   58.851    0.000   69.116    0.000 BlocksWorld.py:156(parse_part_state)
  5649569   24.843    0.000   24.843    0.000 {method 'symbols' of 'clingo.Model' objects}
  2822400   24.463    0.000  109.685    0.000 BlocksWorld.py:181(parse_state)
     4376   20.598    0.005   48.068    0.011 {method 'solve' of 'clingo.Control' objects}
     4376   12.732    0.003   12.732    0.003 {method 'ground' of 'clingo.Control' objects}
      150   12.339    0.082  166.515    1.110 BlocksWorld.py:38(generate_all_states)
 59359167    8.477    0.000    8.477    0.000 entities.py:8(__eq__)
 19786389    7.521    0.000   10.265    0.000 entities.py:2(__init__)
 19786389    3.869    0.000    5.591    0.000 entities.py:14(__hash__)
 45313965    3.357    0.000    3.357    0.000 {method 'append' of 'list' objects}
 19827078    2.753    0.000    2.753    0.000 {method 'replace' of 'str' objects}
      150    2.555    0.017  169.071    1.127 BlocksWorld.py:28(get_random_start_state)
  5649569    2.217    0.000   27.470    0.000 ClingoBridge.py:12(on_model)
     4376    2.208    0.001    2.208    0.001 {method 'load' of 'clingo.Control' objects}
     4226    2.029    0.000   23.904    0.006 BlocksWorld.py:103(next_step)
 19887237    1.735    0.000    1.735    0.000 {built-in method builtins.hash}
    15078    0.931    0.000    0.931    0.000 {method 'add' of 'clingo.Control' objects}
  2826627    0.710    0.000    0.710    0.000 entities.py:38(__init__)
     4376    0.472    0.000    0.472    0.000 ClingoBridge.py:6(__init__)
    54258    0.111    0.000    0.148    0.000 entities.py:53(<listcomp>)
```

Experiment 3b

Planning agent (on empty policy, ph=5) in a 7-blocks world, 150 episodes

Tue Aug 18 17:49:48 2020 exp3b_profile_raw.txt

```
         222659262 function calls (222651560 primitive calls) in 201.846 seconds

   Ordered by: internal time
   List reduced from 2717 to 20 due to restriction <20>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 19793277   53.829    0.000   63.229    0.000 BlocksWorld.py:156(parse_part_state)
     5361   26.016    0.005   52.922    0.010 {method 'solve' of 'clingo.Control' objects}
  5650604   24.328    0.000   24.328    0.000 {method 'symbols' of 'clingo.Model' objects}
  2822400   24.148    0.000  102.977    0.000 BlocksWorld.py:181(parse_state)
     5361   21.068    0.004   21.068    0.004 {method 'ground' of 'clingo.Control' objects}
      150   11.961    0.080  158.370    1.056 BlocksWorld.py:38(generate_all_states)
 59379852    8.277    0.000    8.277    0.000 entities.py:8(__eq__)
 19793284    6.756    0.000    9.400    0.000 entities.py:2(__init__)
 19793284    3.731    0.000    5.405    0.000 entities.py:14(__hash__)
 45385275    3.277    0.000    3.277    0.000 {method 'append' of 'list' objects}
     5211    3.119    0.001   40.129    0.008 BlocksWorld.py:103(next_step)
     5361    2.682    0.001    2.682    0.001 {method 'load' of 'clingo.Control' objects}
 19886164    2.661    0.000    2.661    0.000 {method 'replace' of 'str' objects}
      150    2.442    0.016  160.813    1.072 BlocksWorld.py:28(get_random_start_state)
  5650604    2.185    0.000   26.906    0.000 ClingoBridge.py:12(on_model)
 19903258    1.687    0.000    1.687    0.000 {built-in method builtins.hash}
    17960    1.129    0.000    1.129    0.000 {method 'add' of 'clingo.Control' objects}
  2827612    0.645    0.000    0.645    0.000 entities.py:38(__init__)
     5361    0.542    0.000    0.542    0.000 ClingoBridge.py:6(__init__)
    91638    0.207    0.000    0.258    0.000 BlocksWorld.py:169(parse_action)
```

Experiment 3

Observations

- Both the tabula-rasa and planning-agent take almost equally long!
  - Difference in planning is only ~7 seconds
  - 48 seconds are spent in `solve` function from clingo (vs. 52s for the planner)
  - *Possible culprit*: We compute the optimal path to the goal every step of every period for benchmarking purposes. This is the only (expensive) ASP-procedure that is executed in both the tabula-rasa and planning agent.
- `generate_all_states` is executed exaclty 150 times, and 160s (of 193 in total) is spent in this function. However, states should be generated only once at the very beginning ?!
- If we can trust the time differences, we could reduce the total time by ~48s + ~150s, reducing the total time to ~10s!