SUPPLEMENT for Submission 6580: Parallel Greedy Best-First Search with a Bound on the Number of Expansions Relative to Sequential Search

Primary Keywords: *None*

Abstract

Supplementary material for submission.

S.1 K-Parallel GBFS (KPGBFS)

Algorithm S.1: K-Parallel GBFS (KPGBFS)

$Open \leftarrow \{s_{init}\}$, $Closed \leftarrow \{s_{init}\}$; $\forall i, s_i \leftarrow NULL$
for $i \leftarrow 0, \ldots, k-1$ in parallel **do** ▷ k is the number of the threads
  loop
    while $s_i = NULL$ **do**
      lock($Open$)
      if $\forall j, s_j = NULL$ **then**
        if $Open = \emptyset$ **then** unlock($Open$); **return** $NULL$
      else
        $s_i \leftarrow top(Open)$; $Open \leftarrow Open \setminus \{s_i\}$
      unlock($Open$)
    if $s_i \in S_{goal}$ **then return** s_i
    lock($Open$), lock($Closed$)
    for $s'_i \in succ(s_i)$ **do**
      if $s'_i \notin Closed$ **then**
        $Closed \leftarrow Closed \cup \{s'_i\}$
        $Open \leftarrow Open \cup \{s'_i\}$;
    unlock($Open$), unlock($Closed$)
    $s_i \leftarrow NULL$

S.2 PUHF Pathology with Consistent Heuristics

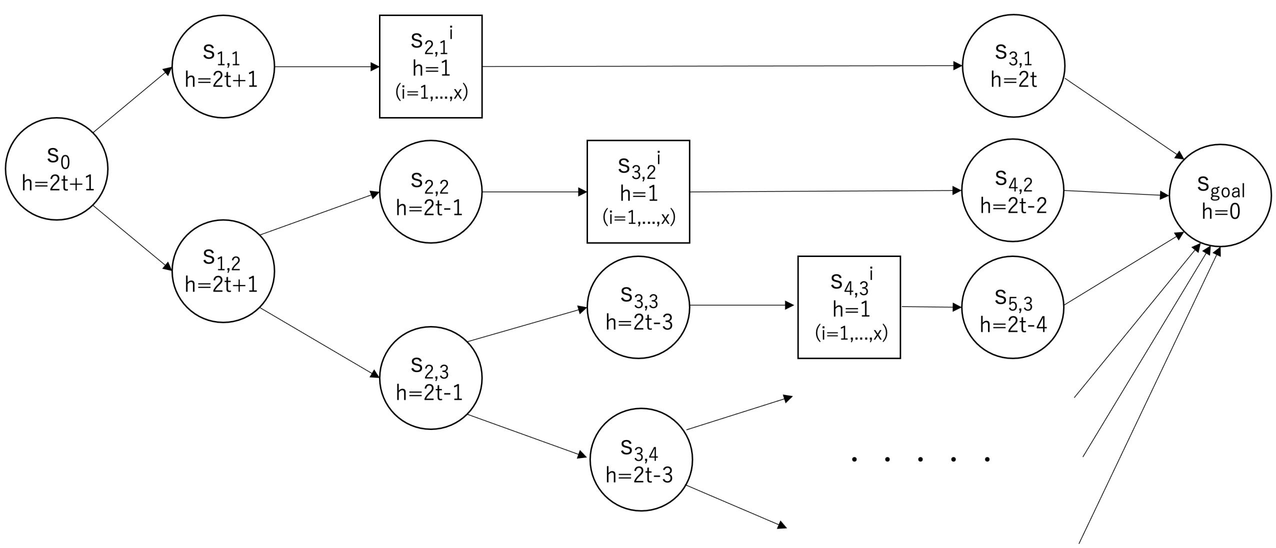

Figure 1: PUHF Pathological Search Behavior Example 2 (same as Figure 2 in the main paper)

PUHF is pathological relative to GBFS even if the heuristic is consistent. For example, consider an instance of the search space with the structure shown in Figure 1, where all edge costs are 1. If we replace all h-values by h', such that $h' = h/c$, where c is a sufficiently large constant such that $h/c < 1$, then h' is consistent, but PUHF is pathological on this search space (by the same argument as the main paper).

S.3 OBAT with SGE (OBAT$_S$)

Algorithm S.2: OBAT with SEE (OBAT$_S$), lock/unlock operations for $Being\_Expanded$ omitted for space

$Open \leftarrow \{s_{init}\}$, $Closed \leftarrow \{s_{init}\}$; $\forall i, s_i \leftarrow NULL$
for $i \leftarrow 0, \ldots, k-1$ in parallel **do**
  loop
    lock($Unevaluated$)
    if $Unevaluated \neq \emptyset$ **then**
      $s_i \leftarrow top(Unevaluated)$
      $Unevaluated \leftarrow Unevaluated \backslash \{s_i\}$
      unlock($Unevaluated$)
      evaluate(s_i) ▷ using a hashtable to prevent reevaluation of states
      $s \leftarrow parent(s_i)$
      if all $succ(s)$ have been evaluated **then**
        if $h(s) \leq min_{s' \in succ(s)} h(s')$ **then** ▷ s is an a-state
          lock($Open$), lock($Closed$)
          for $s' \in succ(s)$ **do**
            if $s' \notin Closed$ **then**
              $Closed \leftarrow Closed \cup \{s'\}$
              $Open \leftarrow Open \cup \{s'\}$
          unlock($Open$), unlock($Closed$)
        else ▷ s is a b-state
          lock($Deferred$)
          $Deferred \leftarrow Deferred \cup \{s\}$
          unlock($Deferred$)
        $Being\_Expanded \leftarrow Being\_Expanded \backslash \{s\}$
    else
      unlock($Unevaluated$)
      lock($Open$), lock($Deferred$)
      if $Open = \emptyset$ and $Deferred = \emptyset$ **then**
        unlock($Open$), unlock($Deferred$)
        if $Being\_Expanded = \emptyset$ **then return** $NULL$
      else
        if $h(top(Deferred)) \leq h(top(Open))$ **then**
          if $h(top(Deferred)) \leq min_{s \in Being\_Expanded} h(s)$ **then**
            $s_i \leftarrow top(Deferred)$; $Deferred \leftarrow Deferred \backslash \{s_i\}$
            $Being\_Expanded \leftarrow Being\_Expanded \cup \{s_i\}$
            lock($Closed$)
            for $s'_i \in succ(s_i)$ **do**
              if $s'_i \notin Closed$ **then**
                $Closed \leftarrow Closed \cup \{s'_i\}$
                $Open \leftarrow Open \cup \{s'_i\}$
            unlock($Closed$)
            $Being\_Expanded \leftarrow Being\_Expanded \backslash \{s_i\}$
          unlock($Open$), unlock($Deferred$)
        else
          if $h(top(Open)) \leq min_{s \in Being\_Expanded} h(s)$ **then**
            $s_i \leftarrow top(Open)$; $Open \leftarrow Open \backslash \{s_i\}$
            $Being\_Expanded \leftarrow Being\_Expanded \cup \{s_i\}$
            unlock($Open$), unlock($Deferred$)
            if $s_i \in S_{goal}$ **then return** $Path(s_i)$
            generate($succ(s_i)$)
            lock($Unevaluated$)
            for $s'_i \in succ(s_i)$ **do**
              $Unevaluated \leftarrow Unevaluated \cup \{s'_i\}$
              $parent(s'_i) \leftarrow s_i$
            unlock($Unevaluated$)
            if $succ(s_i) = \emptyset$ **then** $Being\_Expanded \leftarrow Being\_Expanded \backslash \{s_i\}$
          else
            unlock($Open$), unlock($Deferred$)
    $s_i \leftarrow NULL$

Algorithm S.2 shows OBAT$_S$ (OBAT with SGE).

Although the pseudocode may appear more complex than OBAT, the additional code required for OBAT$_S$ is a straightforward implementation of SGE, very similar to the previous application of SGE to PUHF and KPGBFS (Shimoda and Fukunaga 2024).

Most of the new pseudocode relative to OBAT are in the `if`-block in lines 4–23, which handles the parallel evaluation of states in $Unevaluated$ by available threads.

In OBAT, when a state s leaves $Open$, the thread which removed s generates $succ(s)$, the successors of s, and evaluates the h-values of $succ(s)$.

In OBAT$_S$, when a state s leaves $Open$, the thread which removed s generates $succ(s)$, but instead of evaluating $succ(s)$, inserts $succ(s)$ into $Unevaluated$ (lines 50-54). The unevaluated states in $Unevaluated$ are evaluated and processed as soon as threads become available, having higher priority than removing states from $Deferred$ and $Open$.

$Being\_Expanded$ is an auxiliary data structure which contains the set of states which are currently being expanded according to Definition 10.

S.4 Correctness of OBAT$_S$

Applying SGE to OBAT, KPGBFS, and PUHF can change the search behavior. In the case of KPGBFS, where state expansion is unconstrained and the state with best h-value in the shared $OPEN$ is always expanded, there is no known constraint on the set of states which can be expanded, so adding SGE has no relevant theoretical implication.

In the case of PUHF and its variants, including PUHF3, it was argued in (Shimoda and Fukunaga 2024) that although SGE can change the order in which states are expanded, SGE can not cause PUHF to violate the state expansion constraint, i.e., PUHF with SGE can not expand a state which is not in th BTS. This is because PUHF explicitly checks the state expansion constraint before expanding a state, so it is not possible for PUHF with SGE to expand a state which does not satisfy the expansion constraint.

Now, let us consider OBAT with SGE. Is is clear that like PUHF with SGE, adding SGE to OBAT can not cause the search to violate the two main constraints (1) only states in the BTS are expanded, and (2) only 1 bench is explored at a time, because these constraints are enforced by the `if`-statement conditions in Algorithm S.2, lines 31, 32, 44, and these are the same conditions as in OBAT (Algorithm 1, lines 10, 11, 21).

The proofs for Theorem 2 and Lemma 1 depends only on the expansion constraints and priority ordering scheme for removing states from $Open$ and $Deferred$, which are the same in OBAT and OBAT$_S$, so Theorem 2 and Lemma 1 hold for OBAT$_S$.

It remains to show Lemma 2 (and therefore Theorem 4) holds for OBAT$_S$.

We need to show that the number of states remaining in $Deferred$ after OBAT$_S$ terminates does not exceed $k|p|$, the bound given by Theorem 3.

As OBAT$_S$ differs from OBAT in that SGE separates successor generation and evaluation such that successors are not necessarily immediately evaluated by the same thread which generated them, we define the following:

Definition 10. We say that a state s is *being expanded* if either:

- s has been removed from $Open$, at least one member of $succ(s)$ has not yet been evaluated. or
- s has been removed from $Deferred$, at least one member of $succ(s)$ has not yet been inserted in $Open$.

A thread might remove s from $Open$, insert $succ(s)$ in $Unevaluated$, and move on to remove another state from $Open$ or $Deferred$ while s is still "being expanded" by the definition above. However, we show that there are never more than k states being expanded during OBAT$_S$ search.

Lemma 3. The maximum number of states being expanded by OBAT$_S$ is at most k .

Proof. At any given time, let n be the number of threads which are expanding a state from $Deferred$ (Alg S.2 Lines 33-41) or generating its successors (Alg S.2, Lines 45-49), and let m be the number of states whose successors have been inserted into $Open$, but at least one member of successors has not yet been evaluated.

$n + m$ is the number of states being expanded by OBAT$_S$ at any given time. We show that $n + m \leq k$ by induction.

First, the condition trivially holds at the start of search ($n = 0, m = 0$). Next, we perform induction on n and m. Assume the condition $n + m \leq k$ holds. Consider when n increases: $Unevaluated$ is prioritized over $Open$ and $Deferred$, so $Unevaluated$ is empty. Let l be the number of threads currently evaluating states. Clearly, $m \leq l$. Since $n + l \leq k$, $n + m \leq k$. Consider when m increases: When some thread i inserts a successor state $u \in succ(s_i)$ into $Unevaluated$, m increases by 1. This also means that thread i has finished generating the $succ(s_i)$ so n decreases by 1, and $n + m$ is unchanged, so $n + m \leq k$. This completes the induction. □

This bound on the number of states being expanded allows us to obtain the equivalent of Lemma 2 for OBAT$_S$.

Lemma 4. In OBAT$_S$, for every h-value, $Deferred$ contains at most k states, where k is the number of threads.

Proof. Assume that at some point during search, $k+1$ states which have the same h-value are inserted in $Deferred$. Let $n_1, n_2, \ldots, n_{k+1}$ denote the order in which these states were inserted in $Deferred$.

After n_1 is inserted in $Deferred$, it is not possible to remove $n_i (2 \leq i \leq k+1)$ from $Open$. This is because if $n_i (2 \leq i \leq k+1)$ is removed from $Open$ after n_1 is inserted in $Deferred$, then by the time n_i is removed from $Open$, n_1 must already have been removed from $Deferred$, because among states with the same h-value, removal from $Deferred$ is prioritized over $Open$. This means n_1 and n_i could not both have been simultaneously in $Deferred$, a contradiction.

Thus, immediately before n_1 is inserted in $Deferred$, $n_i (2 \leq i \leq k+1)$ must have been removed from $Open$, and not yet inserted into $Deferred$. i.e., these states are being expanded. Thus, $k+1$ states $(n_1, \ldots, n_{k+1})$ must be simultaneously expanded, but at most k states are being expanded, a contradiction.

□

From Lemmas 1, 3, and 4, Theorem 3 holds for OBAT$_S$, so Theorem 4 also holds for OBAT$_S$.

S.5 Explanation of Benchmark Selection

We compared GBFS, OBAT, OBAT$_S$, KPGBFS, KPGBFS$_S$ (KPGBFS with SGE), PUHF3, PUHF3$_S$ (PUHF3 with SGE) using a set of instances based on the Autoscale-21.11 satisficing benchmark set (42 STRIPS domains, 30 instances/domain, 1260 total instances) (Torralba, Seipp, and Sievers 2021). The Autoscale benchmarks are an improved benchmark suite based on the IPC classical planning benchmarks which were designed to compare the performance of different solvers, as advances in solvers sometimes made performance comparisons among modern solvers difficult using the classic IPC competition benchmark instances.

However, even on the Autoscale-21.11 benchmark set, there were several domains which were too easy – all methods solved all instances, rendering these instances useless for the purpose of comparing coverage among the methods.

Therefore, for domains where (1) all methods solved all instances for $k=4$, $k=8$, and $k=16$ threads, and (2) a parameterized instance generator for the domain is available in the Autoscale repository, we replaced the Autoscale-21.11 instances with more difficult instances generated using the same Autoscale instance generator. Specifically, the domains where criteria (1) and (2) above applied were gripper, and miconic.

- gripper: The Autoscale-21.11 set used values of parameter n from 20 to 165, in increments of 5 (30 instances). Our instances used n from 175 to 465, in increments of 10 (30 instances).
- miconic:

  The Autoscale miconic instances are generated using two parameters, *passengers* and *floors*. In the Autoscale-21.11 set (30 instances), $19 \leq$ *passengers* ≤ 155, where *passengers* monotonically increased by 4 or 5 between consecutive instances, and $11 \leq$ *floors* ≤ 124, monotonically increasing in increments of 3 or 4.

  Similarly, our replacement instances were generated using the following procedure, which calls the Autoscale `generate_instance` function for miconic:

```
passengers=155, floors=124
for i in range(0,90):
     if(i%4==0):
          passengers+=4
     else:
          passengers+=5
     floors+=4
     if(i%3==2):
          generate_instance(passengers,floors)
```

S.6 Coverage Per Domain (full table)

#threads	1 thread	4 threads						8 threads						16 threads					
	GBFS	KPGBFS	KPGBFS$_S$	PUHF3	PUHF3$_S$	OBAT	OBAT$_S$	KPGBFS	KPGBFS$_S$	PUHF3	PUHF3$_S$	OBAT	OBAT$_S$	KPGBFS	KPGBFS$_S$	PUHF3	PUHF3$_S$	OBAT	OBAT$_S$
agricola	26	**30**	**30**	**30**	28	28	**30**	30	30	30	30	30	30	30	30	30	30	30	30
airport	18	**19**	**19**	18	18	**19**	**19**	18	**19**	17	**19**	16	**19**	17	18	17	**19**	15	18
barman	2	3	3	3	3	3	3	**4**	**4**	**4**	**4**	3	**4**	4	4	4	4	4	**5**
blocksworld	6	6	6	**7**	6	6	6	7	6	7	7	**8**	7	7	7	7	**8**	6	7
childsnack	4	3	**5**	3	4	**5**	3	**5**	4	**5**	4	3	3	**5**	3	**5**	4	4	4
data-network	4	**6**	5	5	5	5	5	**7**	5	5	5	4	5	**7**	6	6	4	5	5
depots	4	4	4	4	4	4	4	4	4	4	4	4	4	**5**	**5**	4	4	4	4
driverlog	4	**8**	7	7	7	7	**8**	7	7	7	7	6	**8**	7	8	8	8	7	**9**
elevators	11	15	14	**16**	**16**	15	14	17	17	17	17	16	**18**	18	19	**20**	**20**	**20**	19
floortile	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
freecell	20	21	21	**24**	21	23	22	22	**23**	21	**23**	22	**23**	24	23	21	22	22	**25**
ged	7	8	8	7	**9**	**9**	**9**	9	8	8	9	**10**	9	9	9	7	8	**10**	8
grid	5	**6**	5	5	**6**	5	**6**	5	5	**6**	**6**	**6**	**6**	5	**6**	**6**	**6**	**6**	**6**
gripper (replaced)	2	4	**10**	4	**10**	**10**	**10**	7	**15**	7	**15**	14	**15**	7	**21**	7	20	19	20
hiking	4	**7**	**7**	**7**	6	6	6	**10**	7	8	7	6	7	**14**	9	11	8	10	9
logistics	4	5	5	5	**6**	5	5	6	6	6	6	6	6	**6**	**6**	5	5	**6**	**6**
miconic (replaced)	10	9	13	10	13	10	**15**	9	15	9	15	10	**18**	9	15	9	15	10	**21**
mprime	4	5	**6**	4	4	4	5	5	**6**	**6**	5	5	5	**6**	**6**	**6**	**6**	5	5
nomystery	6	**11**	8	9	9	5	7	**14**	9	10	8	4	7	**14**	11	11	11	3	6
openstacks	7	9	9	9	9	15	**17**	11	9	10	10	15	**20**	12	11	11	11	15	**22**
organic-synthesis-split	9	16	**17**	16	16	12	16	**17**	**17**	16	**17**	13	**17**	**17**	**17**	16	16	13	16
parcprinter	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
parking	6	**8**	7	6	6	7	6	7	**8**	**8**	**8**	**8**	**8**	9	**10**	9	9	9	**10**
pathways	11	12	12	12	**13**	11	12	11	**15**	12	13	10	9	**15**	12	10	12	11	12
pegsol	30	30	30	30	30	30	30	**30**	**30**	**30**	**30**	**30**	29	30	30	30	30	30	30
pipesworld-notankage	8	8	7	**9**	8	**9**	8	6	8	**10**	9	**10**	7	**11**	9	9	9	9	**11**
pipesworld-tankage	9	11	**12**	**12**	**12**	10	10	10	**12**	**12**	**12**	10	10	**12**	**12**	11	11	**12**	**12**
rovers	26	26	26	26	26	26	26	**27**	25	26	25	26	26	**27**	**27**	26	26	26	26
satellite	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
scanalyzer	7	9	9	9	9	9	9	10	10	10	10	10	10	11	11	11	11	11	11
snake	7	9	9	9	9	9	9	9	9	9	9	9	9	**10**	**10**	9	9	9	9
sokoban	16	18	**19**	17	18	18	18	21	**22**	19	18	19	19	**23**	22	21	21	19	19
storage	3	2	3	3	3	3	**4**	3	3	**4**	3	**4**	**4**	2	**4**	**4**	**4**	3	**4**
termes	12	15	**16**	**16**	**16**	14	**16**	**18**	**18**	16	17	15	16	**18**	**18**	16	**18**	15	16
tetris	8	9	9	9	9	**10**	9	11	11	11	11	11	**12**	**13**	12	**13**	12	**13**	**13**
thoughtful	14	**18**	**18**	**18**	16	16	15	**18**	17	16	17	17	17	**21**	18	19	16	16	14
tidybot	12	11	12	12	12	12	**14**	12	11	11	12	13	**15**	**14**	13	12	11	**14**	13
tpp	8	9	9	9	9	9	9	9	9	9	9	**10**	**10**	10	11	9	10	11	**12**
transport	5	7	7	7	7	7	7	7	**8**	7	7	7	7	**8**	7	**8**	7	7	7
visitall	14	15	15	12	15	12	**16**	15	**18**	14	16	15	16	**20**	19	13	14	15	16
woodworking	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
zenotravel	7	9	9	9	9	9	9	9	9	9	9	**11**	10	11	**12**	**12**	10	11	11
Sum(1260)	401	462	472	459	468	458	**478**	488	500	477	494	477	**506**	529	**532**	494	510	496	**532**

Table 1: Coverage (number of problems solved out of 1260) on Autoscale-21.11 IPC-based planning benchmark set (gripper, and miconic are replaced with harder instances. See S.5).

S.7 Comparisons Without Separate Generation and Evaluation (SGE)

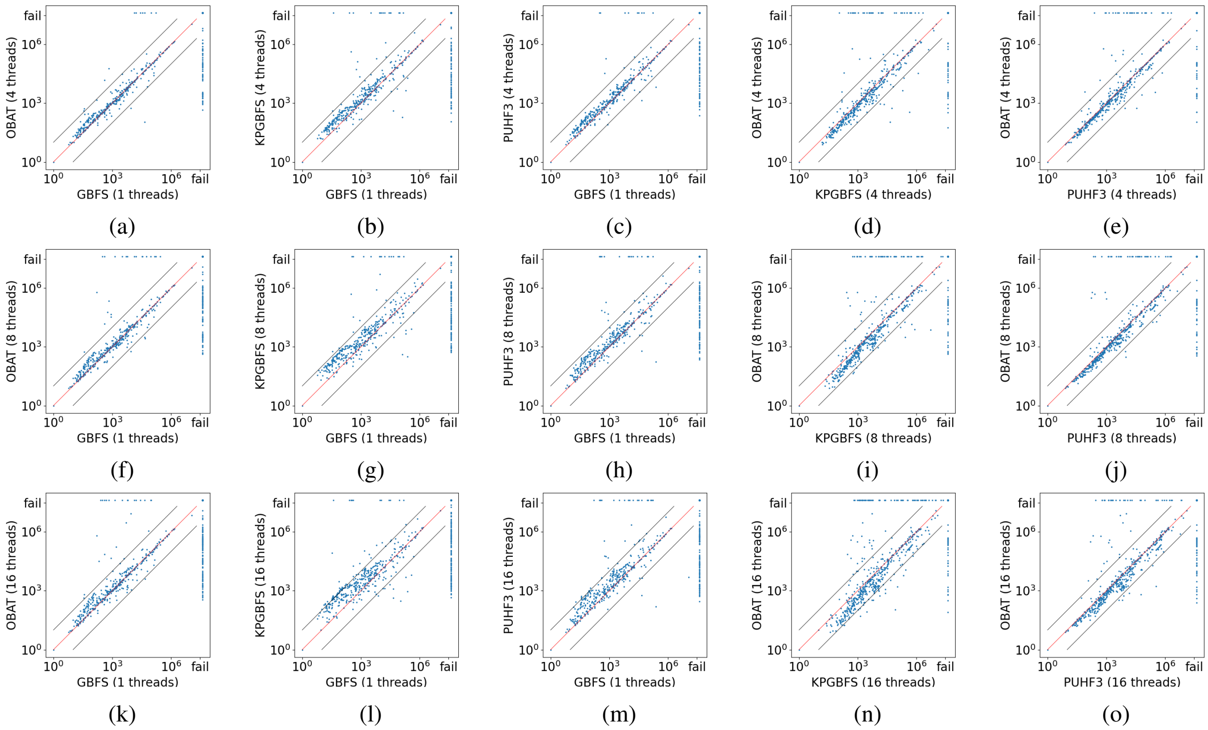

Figure 2: Number of states expanded, Diagonal lines are $y = 0.1x$, $y = x$, and $y = 10x$

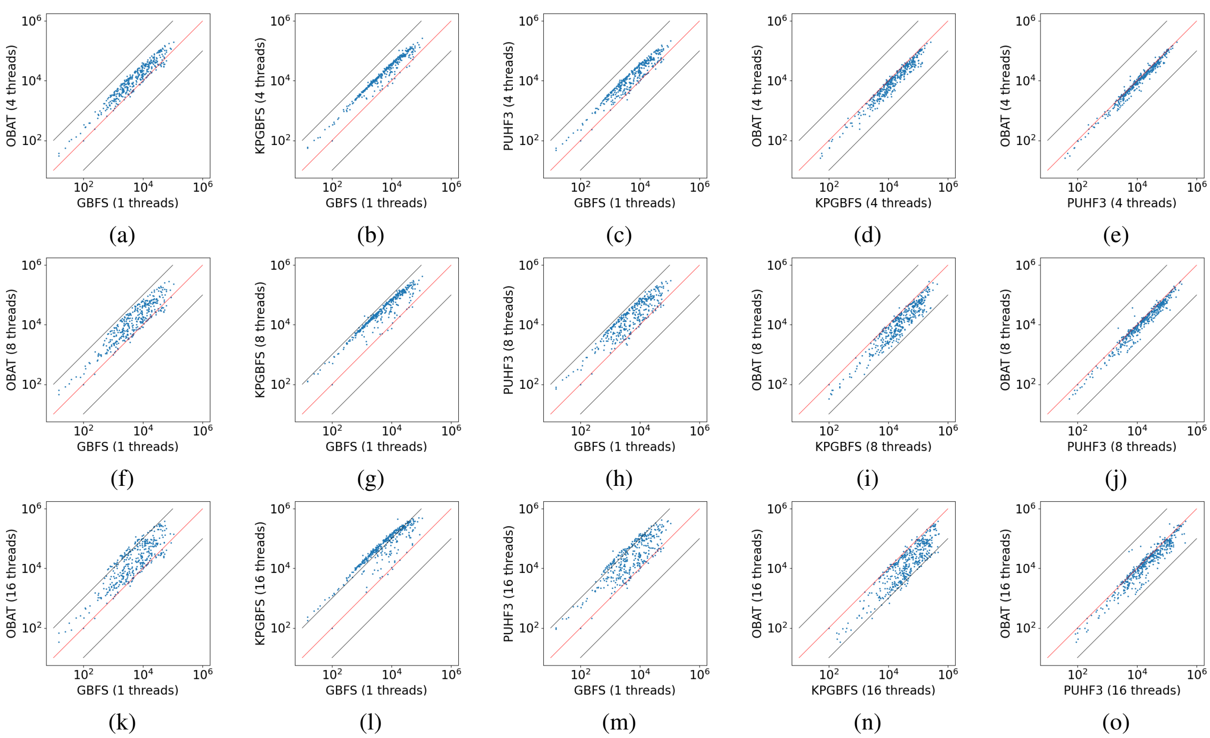

Figure 3: State evaluation rate comparison (states/second), Diagonal lines are $y = 0.1x$, $y = x$, and $y = 10x$

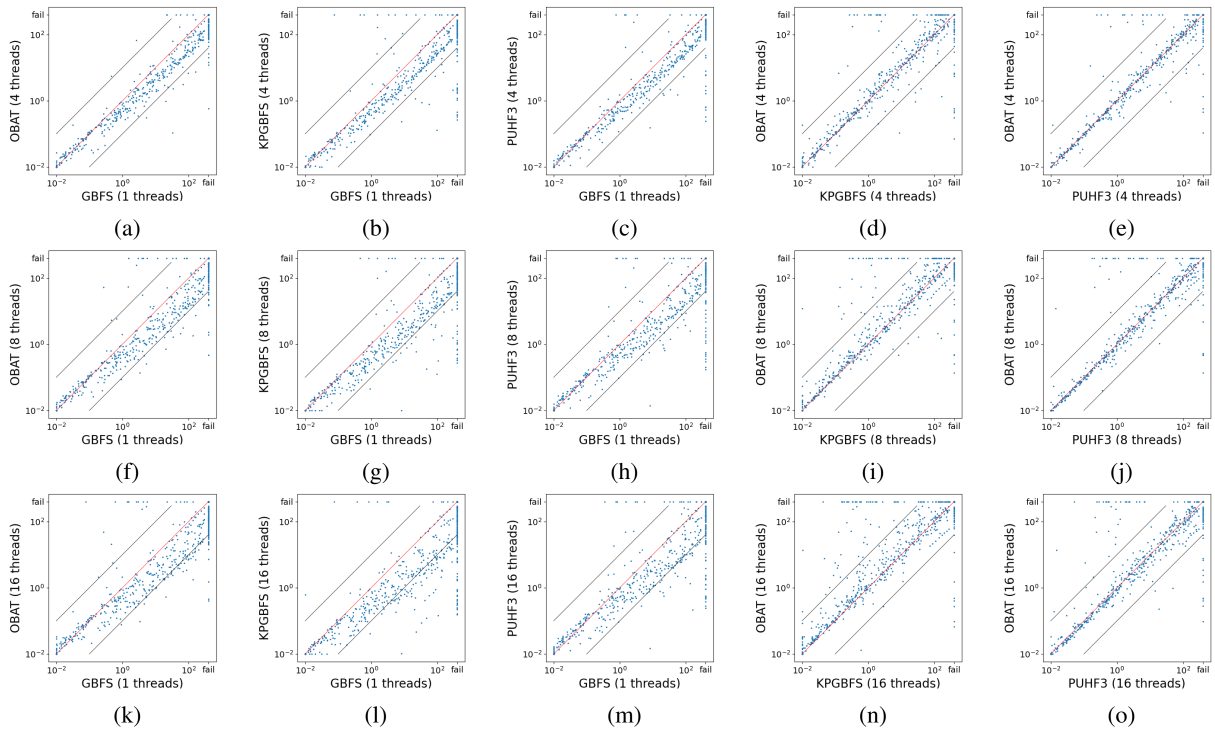

Figure 4: Search time (seconds) “fail”= out of time/memory, diagonal lines are $y = 0.1x$, $y = x$, and $y = 10x$

S.8 Comparisons Including Separate Generation and Evaluation (SGE)

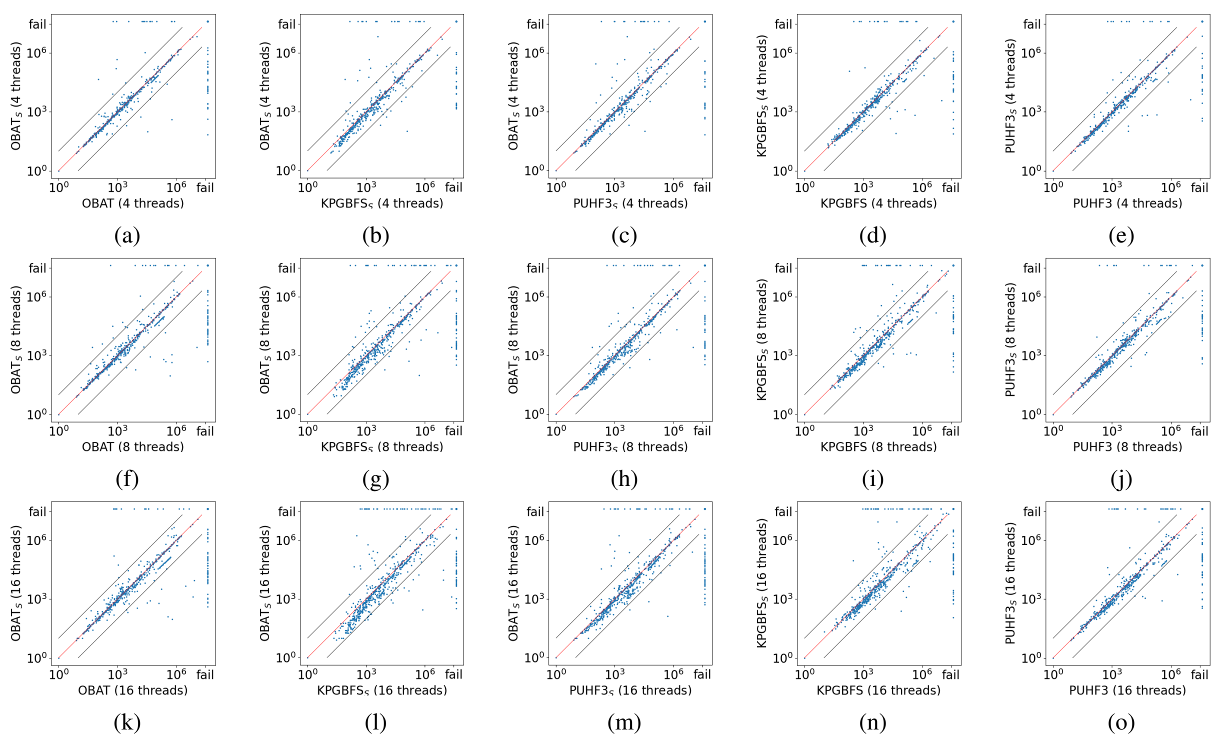

Figure 5: Number of states expanded, Diagonal lines are $y = 0.1x$, $y = x$, and $y = 10x$

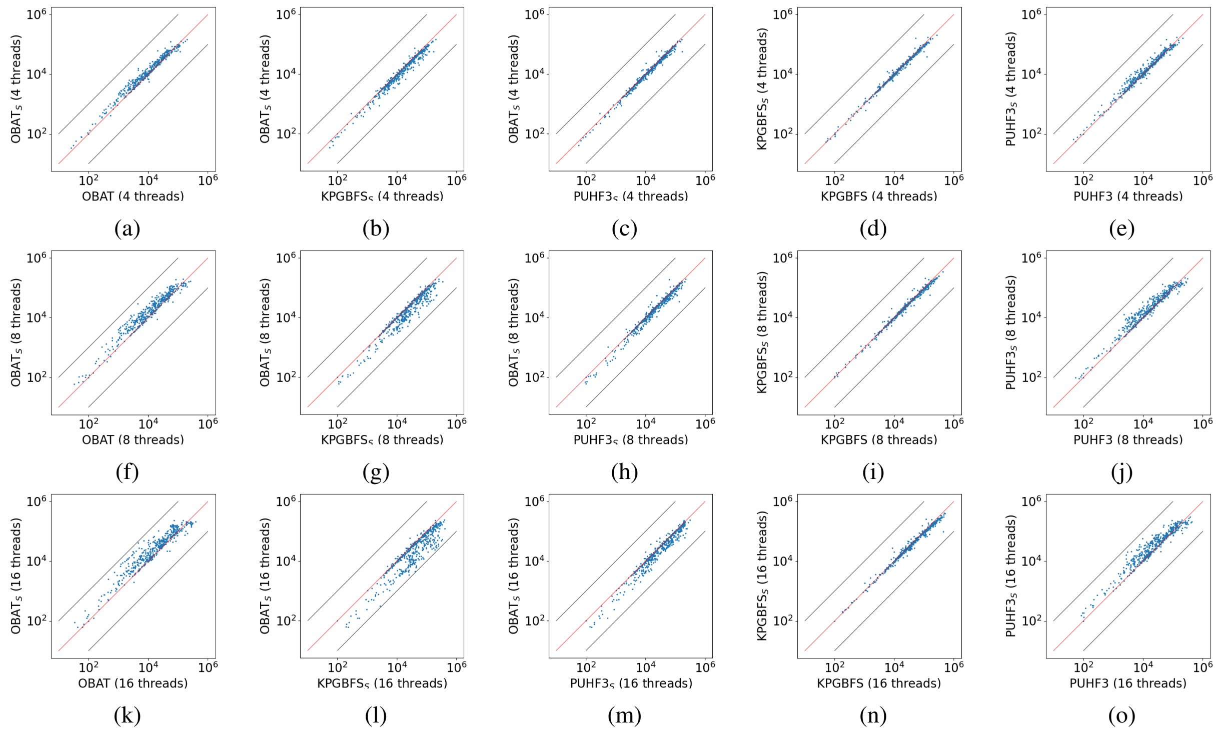

Figure 6: State evaluation rate comparison (states/second), Diagonal lines are $y = 0.1x$, $y = x$, and $y = 10x$

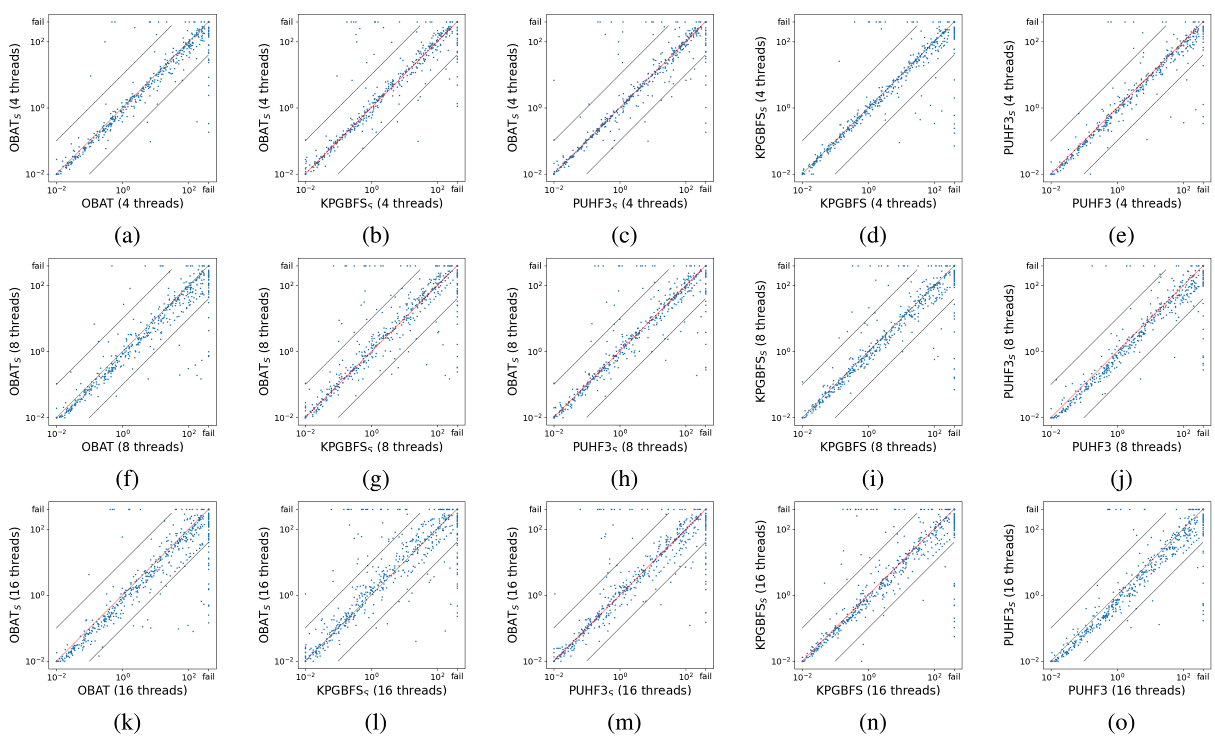

Figure 7: Search time (seconds) “fail”= out of time/memory, diagonal lines are $y = 0.1x$, $y = x$, and $y = 10x$

S.9 The performance impact of SGE on KPGBFS

Our experimental results show that KPGBFS$_S$ (KPGBFS with SGE) had higher overall coverage (472/500/532 for 4/8/16 threads) than KPGBFS (462/488/529 for 4/8/16 threads).

In contrast, (Shimoda and Fukunaga 2024), the experimental evaluation of SGE on KPGBFS showed that KPGBFS$_S$ had slightly lower total coverage (507/529/565 for 4/8/16 threads) than KPGBFS (510/534/567 for 4/8/16 threads).

These results are *not* contradictory – the difference is due to the fact that Shimoda and Fukunaga (2024) used the unmodified version of the AutoScale-21.11 benchmark set, while we used a modified version of the AutoScale-21.11 benchmark set. As explained in Section 7 and Supplement Section S.5, we replaced gripper and miconic with harder instances as all 30 of the original instances were solved by all methods and therefore unsuitable (too easy) for a performance evaluation.

If we remove the new gripper and miconic instances from the total coverage, and then replace them with the much easier, original, AutoScale-21.11 gripper and miconic instances used by (Shimoda and Fukunaga 2024), the total coverage for KPGBFS$_S$ is 509/530/556 for 4/8/16 threads, and total coverage for KPGBFS is 509/532/573 for 4/8/16 threads, which is similar to the results in (Shimoda and Fukunaga 2024).