On the Pathological Search Behavior of Distributed Greedy Best-First Search

Ryo Kuroiwa, Alex Fukunaga
Graduate School of Arts and Sciences
The University of Tokyo

Abstract

Although A* search can be efficiently parallelized using methods such as Hash-Distributed A* (HDA*), distributed parallelization of Greedy Best First Search (GBFS), a suboptimal search which often finds solutions much faster than A*, has received little attention. We show that surprisingly, HDGBFS, an adaptation of HDA* to GBFS, often performs significantly worse than sequential GBFS. We analyze and explain this performance degradation, and propose a novel method for distributed parallelization of GBFS, which significantly outperforms HDGBFS.

1 Introduction

Greedy Best First Search (GBFS) (Doran and Michie 1966) is a heuristic search algorithm which is widely used for quickly finding solutions to difficult graph search problems. GBFS is a best-first search strategy similar to A* (Hart, Nilsson, and Raphael 1968), but unlike A*, which selects nodes for expansion according to the evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach n from the start node and $h(n)$ is an estimate of the cost to reach the closest goal node from n, GBFS selects nodes for expansion according to $f(n) = h(n)$. Thus, GBFS relies entirely on the heuristic estimate $h(n)$, and although it often finds solutions faster than A*, GBFS is a satisficing search algorithm which may return suboptimal solutions. GBFS is widely used for satisficing planning (Richter and Westphal 2010; Xie, Müller, and Holte 2014; Asai and Fukunaga 2017).

Since best-first search strategies such as A* and GBFS may require vast amounts of time and memory to solve difficult problems, parallelization is necessary to efficiently exploit available resources. In particular, parallelization of best-first search across multiple machines is important because for many problems, the bottleneck for best-first search is RAM – best-first search terminates with failure when the available RAM is exhausted. By using the aggregate RAM on multiple machines, a distributed parallel best-first search can solve some problem instances which are not solvable on a single machine.

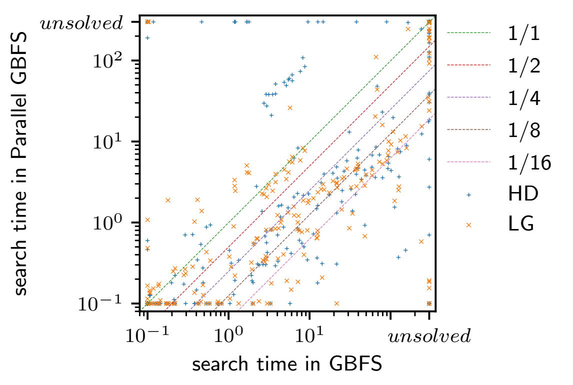
For optimal (admissible) search, an effective method for parallel distributed search is Hash-Distributed A* (HDA*) (Kishimoto, Fukunaga, and Botea 2013), in which each process executes A* with its own local OPEN/CLOSED lists, and all expanded nodes are sent (assigned) to its unique owner, according to a global hash function. HDA* has been shown to scale effectively to over two thousand processes.

Previous work on non-admissible best-first search has focused on multi-core (single machine) parallelizations, including the Weighted A* (Pohl 1970) variant of Parallel Best-Nblock First (PBNF) (Burns et al. 2010), and an implementation of K-Best First Search (KBFS) (Felner, Kraus, and Korf 2003) on a multi-core machine (Vidal, Bordeaux, and Hamadi 2010). To our knowledge, parallel GBFS variants which are suited for both multi-core and multi-machine clusters have not previously been evaluated in-depth.

In this paper, we first evaluate Hash Distributed GBFS (HDGBFS), a straightforward adaptation of HDA* to GBFS, and show that it performs much worse than GBFS on many instances. We show that unlike HDA*, which behaves similarly to A*, HDGBFS behaves very differently from GBFS, searching many regions which are not explored at all by GBFS. We propose LG, a novel variant of HDGBFS which seeks to avoid this performance degradation by forcing each processor to explore highly promising paths locally instead of always sending nodes to their hash-assigned owner. We show that LG significantly outperforms HDGBFS and is competitive with and complementary to a parallel greedy portfolio.

2 Experimental Analysis of HDGBFS

HDA* is a parallelization of A* which distributes work among n processors according to a global hash function. Each process i has local $OPEN_i$ and $CLOSED_i$ lists, and asynchronously executes a cycle in which s, the lowest f-value state in $OPEN_i$ is expanded. Then, for every successor s' of s, $H(s')$, the global hash value of s' is computed, and s' is sent to process $j = H(s') \bmod n$ and put in $OPEN_j$. In order to guarantee optimality, when a process i finds a solution, search continues until a termination protocol determines that there exists no state with the cost less than that of i.



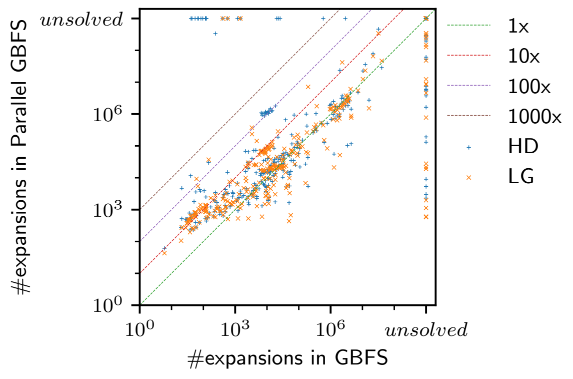


Figure 1: Search time and #expansions in GBFS/h_{lmc} vs. HDGBFS/h_{lmc} (HD) and LG/h_{lmc} (LG). 5min, 128GB for GBFS, 8GB/core for HD and LG.

Hash-based work distribution can be directly applied to GBFS. In *Hash Distributed GBFS (HDGBFS)*, process i selects the minimal h-value state from $OPEN_i$ for expansion. Unlike HDA*, HDGBFS does not guarantee optimality, so it terminates immediately when a solution is found.

We evaluated the performance of HDGBFS by comparing its performance with GBFS. As with standard HDA*, we used the Zobrist hash function (Zobrist 1970) to assign states to processors. We used a FIFO (First In First Out) tie-breaking policy. All of our algorithms were implemented using C++11 and OpenMPI 3.1.1. We used the MPI_Bsend (buffer size 1GB/process), MPI_Iprob and MPI_Recv functions for communications. We used 20 benchmark domains from the satisficing track of IPC-11 and IPC-14 (20 instances/domain). Domains with conditional effects were excluded. For the domains with overlaps in IPC-11 and IPC-14, the IPC-14 version was used.

We used the Landmark Count (LMC) heuristic (Hoffmann, Porteous, and Sebastia 2004). h_{lmc} can be computed quickly and allows a much faster node generation rate than other heuristics such as h_{ff} (Hoffmann and Nebel 2001), so HDGBFS with h_{lmc} is a more interesting platform to evaluate parallel implementation issues than HDGBFS with a slower heuristic like h_{ff}. Results for h_{ff} are qualitatively similar to those of h_{lmc}; we show node expansion comparison for h_{ff} in Fig. 2.

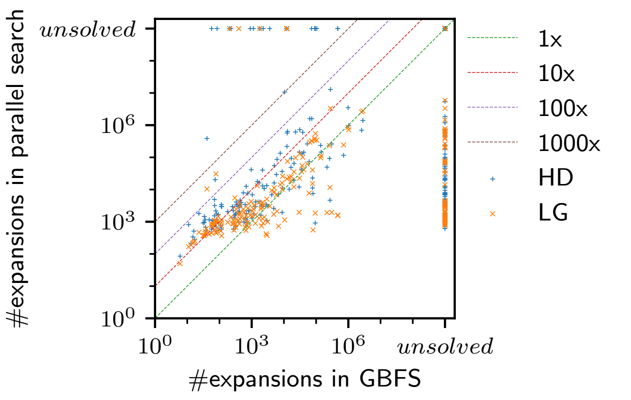


Figure 2: #expansions in GBFS/h_{ff} vs. HDGBFS/h_{ff} (HD) and LG/h_{ff} (LG). 5min, 128GB for GBFS, 8GB/core for HD and LG.

We ran HDGBFS on a machine with 16 cores (Xeon E5-2650 v2 2.60 GHz server with 128GB RAM), and compared vs. GBFS on the same machine. We use a wall-clock time limit of 5 min and a 128GB total RAM usage limit for both GBFS and HDGBFS, i.e., we compare GBFS with 128GB vs. HDGBFS with 8GB RAM/core (128GB total).

Fig. 1 compares HDGBFS vs. GBFS with respect to search time and the number of nodes expanded. Each blue point represents 1 instance. Instances which are unsolved by a method are shown with search time and the number of nodes expanded as "unsolved". Instances solved within 0.1 seconds are shown with 0.1 seconds. Although HDGBFS achieves speedups vs. GBFS on some instances, there are many instances where HDGBFS is much slower than GBFS, and in most cases, HDGBFS expands many more nodes than GBFS. In particular, note that there are many instances which are solved very quickly (1-10 seconds, $< 10^3$ expansions) by GBFS on which HDGBFS either fails or requires dramatically more search effort.

Table 1 evaluates parallel efficiency per domain. Specific domains where HDGBFS performed poorly include nomystery, parcprinter, maintenance, openstacks, thoughtful, and visitall. In maintenance, all 10 instances solved by GBFS (and not by HDGBFS) were solved by GBFS in <2 seconds and 150 expansions. On the other hand, HDGBFS achieves super-linear (more than 16 times) speedup and negative search overhead on parcpritner, pegsol, and transport. These results suggest that HDGBFS behaves very differently from GBFS on some instances.

In total, HDGBFS failed to solve 22/272 instances solved by GBFS, and has lower overall coverage than GBFS, even if we count the 18 instances solved by HDGBFS but not by GBFS. Although HDGBFS achieves speedup on some instances, there are many cases of severe performance degradation on many instances which are easily solved by GBFS.

		HDGBFS				LE				LG			
	base	+	-	spd	so	+	-	spd	so	+	-	spd	so
elevators	7	1	0	1.76	3.08	1	0	8.32	0.75	**2**	0	10.14	0.77
nomystery	13	1	2	0.00	6553.98	2	2	0.07	103.01	**4**	**0**	1.22	0.09
parcprinter	14	**2**	4	20.56	-0.54	1	**0**	44.66	-0.66	0	**0**	15.93	-0.39
pegsol	20	0	0	38.70	-0.81	0	0	92.14	-0.95	0	0	45.66	-0.90
scanalyzer	20	0	0	0.37	0.06	0	0	0.28	-0.28	0	0	0.26	-0.12
sokoban	9	5	0	1.91	4.47	5	0	3.73	1.27	5	0	2.18	3.28
tidybot	19	0	**0**	5.30	1.73	0	1	7.26	1.17	0	**0**	9.13	0.77
woodworking	4	**5**	0	10.45	0.19	3	0	10.61	0.14	4	0	16.51	-0.29
barman	20	0	0	13.43	-0.07	0	0	12.51	-0.07	0	0	6.41	0.77
floortile	1	1	0	1.42	2.64	1	0	2.03	1.41	1	0	1.56	2.04
ged	20	0	0	2.83	0.62	0	0	3.17	0.54	0	0	2.23	1.19
hiking	20	0	0	7.35	0.27	0	0	2.97	2.24	0	0	3.02	2.09
maintenance	14	0	10	0.82	12.99	**1**	7	0.79	10.00	**1**	**0**	0.40	10.25
openstacks	20	0	3	3.90	0.78	0	**0**	3.87	1.09	0	**0**	5.68	0.00
tetris	20	0	0	12.18	-0.32	0	0	11.87	-0.32	0	0	12.01	-0.28
thoughtful	15	**3**	3	5.28	0.29	**3**	**1**	3.06	0.32	2	4	4.98	0.31
transport	16	0	0	20.42	-0.49	0	0	16.77	-0.49	0	0	18.30	-0.50
visitall	20	0	0	0.09	103.50	0	0	0.86	9.15	0	0	1.14	6.93
total	272	18	22	-	-	17	11	-	-	**19**	**4**	-	-

Table 1: Coverage, speedup, and search overhead of parallel GBFS per domain. Speedup (spd) is defined as time(GBFS)/time(Parallel GBFS). Search Overhead (so) is defined as (#expansions(Parallel GBFS)-#expansions(GBFS))/#expansions(GBFS). spd/so in each domain is computed from the sum of time/#expansions on instances solved by both GBFS and Parallel GBFS. Coverage is shown as follows: 'base' is the number of instances solved by GBFS, '+' is the number of instances solved by Parallel GBFS but not solved by GBFS, and '-' is the number of instances solved by GBFS but not solved by Parallel GBFS.

2.1 Communications Overhead and HDGBFS

One factor responsible for the poor performance of HDGBFS is communications overhead. HDGBFS sends all generated nodes to their owner, where they are evaluated and then placed in the owner's OPEN list. If node evaluation is relatively expensive, nodes may be stuck in the owner's receive buffer for a long time before being evaluated, or the receive buffer may overflow. In the openstacks domain (large # of successors/node), HDGBFS crashed on several instances due to overflowing the 1GB MPI_Bsend buffer.

To address this issue, we implemented LE (Local Evaluation), which evaluates nodes at the processor where they are generated, rather than at the receiver (owner) node as in HDGBFS. Each process avoids unnecessary evaluation by performing duplicate detection using its own CLOSED list before the evaluation. Although this may seem like a minor implementation detail, in some domains such as visitall, shifting the cost of evaluation to the generating processor prevents the bottleneck described above. This significantly improves performance on parcprinter, openstacks, and visitall as shown in Table 1. However, overall, LE still performs poorly on some domains compared to GBFS.

We also tried other methods for improving communications overhead, including packing multiple states per message (Kishimoto, Fukunaga, and Botea 2013) and abstract Zobrist hashing (Jinnai and Fukunaga 2017), but failed to get significant overall improvements.

To evaluate the extent to which communications could be responsible for performance degradation in HDGBFS, we consider an ideal model which eliminates all communications overhead related issues. KGBFS is a deterministic, sequential adaptation of k-best first search (KBFS) (Felner, Kraus, and Korf 2003) to GBFS. At each iteration, KGBFS removes k nodes with the minimal h-value from OPEN, and inserts their successors into OPEN.

Under the 3 following conditions, HDGBFS with n processes expands states in the same order as KGBFS with $k = n$. (1) All processes are synchronized so that they all perform each local expansion simultaneously. (2) The hash function H used by HDGBFS is ideal, such that at each expansion step, the node with the (global) i-th smallest h-value is in $OPEN_i$. (3) Communications are instantaneous.

The expansions are synchronized across all processors (assumption 1), and at each step, the node with the global i-th smallest h-value is expanded by the i-th process (assumption 2), and their successors are instantaneously sent to their owner processes so that they are available for expansion in the next step (assumption 3). Thus, although the OPEN list is distributed, the global behavior of this algorithm is identical to that of KGBFS with $k = n$

We evaluated KGBFS with $k = 16$, and there were 3 instances in thoughtful, 1 instance in maintenance, and 1 instance in nomystery which were solved by GBFS within 1 second with #expansions < 100 (the nomystery maintenance instances), and #expansions < 1000 (thoughtful), but were not solved by KGBFS within a 5 minute time limit – even with an idealized communications model, KGBFS often performs much worse than GBFS.

Thus, we have shown that although communications related overheads are partially responsible for the poor performance of HDGBFS, there appears to be a deeper issue responsible for the poor performance of HDGBFS.

3 GBFS vs. HDGBFS Search Behavior

As shown in the previous section, there are some domains where HDGBFS expands 10 - 10000 times as many nodes as GBFS, resulting in severe performance degradation. Although communications overhead is partially related, it does not fully explain why the problem occurs.

In HDA*, three causes of search overhead have been identified (Jinnai and Fukunaga 2017): 1) Band Effect: The expansion order of HDA* differs from that of A* due to f-value imbalance among processors; 2) Burst Effect: HDA* expands states with high f-value because OPEN lists are empty at the beginning of the search; and 3) Node reexpansions. In HDGBFS, h-value imbalance like Band Effect and expansion of states with high h-value like Burst Effect are possible. Node reexpansion is irrelevant because HDGBFS does not reexpand states. A* with a consistent heuristic never expands any state with $f > f^*$, but in HDA*, the band and burst effects sometimes increase expansion of states with $f = f*$ and cause expansion of states with $f > f*$. However, the burst effect is a brief, temporary phenomenon (Jinnai and Fukunaga 2017) and cannot explain very large search overhead on difficult problems.

To test whether the band effect can be observed in HDGBFS, we plotted the node expansion orders of GBFS vs. HDGBFS for barman-p1-11-4-15 and maintenance-1-3-060-180-5-001 in Fig. 3.

In the case of barman-p1-11-4-15, where HDGBFS was successful (speedup=13.43, so=-0.07), the expansion

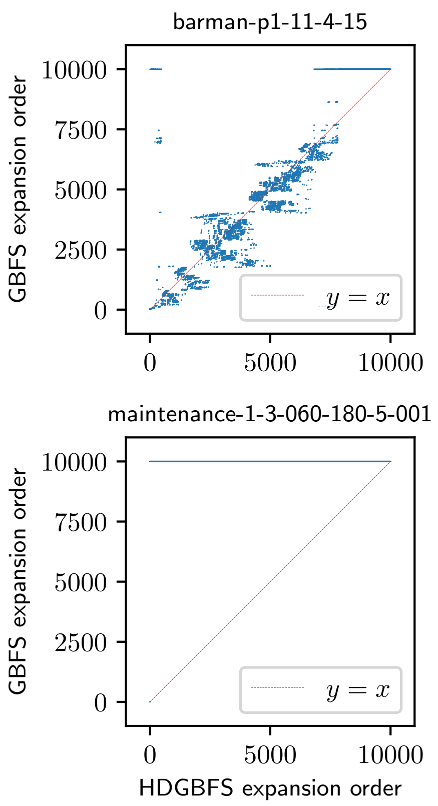


Figure 3: GBFS/h_{lmc} with 10000 expansion limit (but search was continued after a goal was found) vs. HDGBFS/h_{lmc} (5min, 4 processes) expansion order. Nodes expanded by HDGBFS but not expanded by GBFS after 10000 expansions are plotted at $y = 10000$.

orders of HDGBFS and GBFS are similar, and something resembling a band effect could be observed. On the other hand, on maintenance-1-3-060-180-5-001, where HDGBFS fails (not solved after 5 min.), most of the points are at y=10000, i.e., HDGBFS ends up searching an entirely different region of the search space than GBFS.

Thus, the previously identified sources of search overhead in HDA* are not sufficient to explain the large search overhead observed for HDGBFS in cases such as maintenance-1-3-060-180-5-001 – another explanation based on the search behavior of GBFS is necessary.

We analyzed part of the maintenance-1-3-060-180-5-001 search space. GBFS finds the goal after 40 expansions – at every step, the h-value of the best successor is an improvement over the parent, and choosing the lowest h-value with FIFO tie-breaking leads directly to the goal, so the search simply follows a straight path from the start state to the goal. The first 151 states expanded by HDGBFS (4 processes) is shown in Fig. 3. Unlike GBFS, HDGBFS explores multiple regions of the space in parallel, and although HDGBFS initially follows a straight path, after the 75th node expansion, the search explodes. The search space continues to expand after #152 (not shown due to space). Most of the nodes after #100 have the same h-value of 1, i.e., HDGBFS enters a

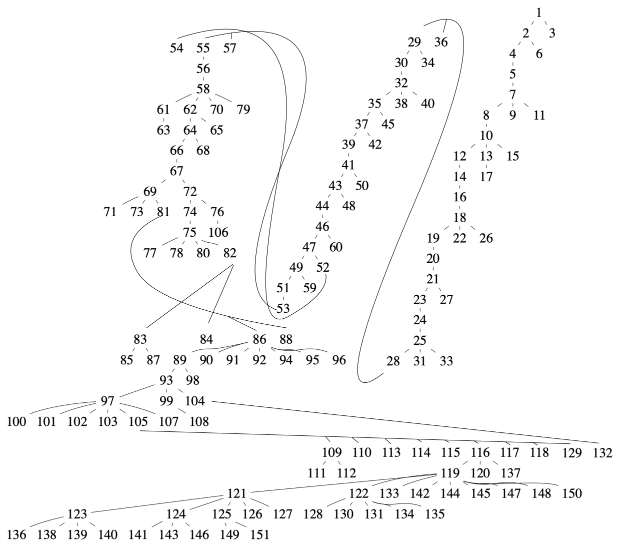


Figure 4: The first 151 nodes expanded by HDGBFS (4 processes) on maintenance-1-3-060-180-5-001. Node IDs=expansion order, edges represent parent-successors.

plateau region. Thus, HDGBFS clearly searches the search space in a different way than GBFS, and causing HDGBFS to fail to solve a very easy problem within the 5-min limit.

Since HDGBFS and GBFS frequently explore entirely different regions of the search space, analysis of HDGBFS search overhead based on expansion order, which was an effective tool for HDA* (Jinnai and Fukunaga 2017), is too coarse-grained – we need a different tool for quantitatively comparing the search behaviors of parallel GBFS.

3.1 Benches and HDGBFS/KGBFS

Next, we theoretically and empirically investigate the cause of this search overhead by applying the recently proposed notions of high-water marks and benches (Wilt and Ruml 2014; Heusner, Keller, and Helmert 2017). Definitions and properties of the high-water marks and benches are from (Heusner, Keller, and Helmert 2017).

A state space $\mathcal{S} = \langle s_I, S_*, succ, cost\rangle$ is defined by an initial state s_I, a set of goal states S_*, a successor function $succ$, and a cost function $cost(s, s')$, $s' \in succ(s)$. For state space $\mathcal{S}$, S is the minimal set of states which satisfy $s_I \in S, S_* \subseteq S, succ(s) \subseteq S$ $(\forall s \in S)$. We call $\rho = \langle s_0, ...s_n\rangle$, a path from s_0 to s_n when $s_i \in succ(s_{i-1})$ $(\forall i = 1, ..., n)$. We represent the set of solutions from s, i.e. $\{\langle s_0, ..., s_n\rangle | s_i \in succ(s_{i-1})$ $(\forall i = 1, ..., n), s_0 = s, s_n \in S_*\}$ as $P(s)$. The cost of path ρ, $cost(\rho) = \sum_{i=0}^{n-1} cost(s_i, s_{i+1})$. A state space topology $\langle\mathcal{S}, h\rangle$ is a tuple consisting of a state space and a heuristic function $h : S \to \mathbb{R}_0^+$, where $h(s)$, the heuristic value of state s is an estimate of the cost to reach a goal state from s.

The high-water mark of a state space topology is defined as follows:

Definition 1 *Let $\langle\mathcal{S}, h\rangle$ be a state space topology, and let $s \in S$ be a state. The* high-water mark *of s is defined as*

$$hw_h(s) := \begin{cases} \min_{\rho \in P(s)}(\max_{s' \in \rho} h(s')) & \text{if } P(s) \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$

The high water-mark of a set of states $S' \subseteq S$ is

$$hw_h(S') := \min_{s \in S'} hw_h(s)$$

A high-water mark bench is defined as follows:

Definition 2 *Let $\langle S, h \rangle$ be a state space topology with set of states S, and let $S' \subseteq S$. The* high-water mark bench $\mathcal{B}_h(S')$ *of S' is a 3-tuple $\langle I, B, E \rangle$ with $B \subseteq S, I \subseteq B$, and $E \subseteq B$. $\mathcal{B}_h(S')$ is defined as follows: if S' contains a goal state, then $\mathcal{B}_h(S') = \langle \emptyset, \emptyset, \emptyset \rangle$.*

Otherwise, let all states $s \in S$ with $h(s) \leq hw_h(S'), hw_h(s) \geq hw_h(S')$, and $s \notin S_*$ be the bench state candidate *set, and define the* bench states *B as the set of all states that can be reached from some state in S' on some path that only includes states from the candidate set; the set of bench* entry states *as $I = S' \cap B$; and the set of bench* exit states *as $E = \{s \in B | hw_h(succ(s)) < hw_h(S')$ or $succ(s) \cap S_* \neq \emptyset\}$. The high-water mark of the bench states B of a bench $\mathcal{B}_h$ is abbreviated to $hw_h(\mathcal{B}_h)$, and $s \in \mathcal{B}_h$ means $s \in B(\mathcal{B}_h)$.*

For brevity, we use "bench" to mean a high-water mark bench.

After GBFS expands a bench exit state, no more states in that bench will be expanded. GBFS first explores $\mathcal{B}_h(\{s_I\})$. After expanding bench exit state s, GBFS continues to expand only the states in $\mathcal{B}_h(succ(s))$, until it exits that bench. This is repeated until a goal is found. Thus, GBFS explores 1 bench at a time until a goal state is found. A *bench path* is the sequence of benches explored by GBFS from $\mathcal{B}_h(\{s_I\})$ to a goal.

When there are multiple paths to goals (e.g., in domains with symmetry), it is possible for a bench to have multiple bench exit states leading to different benches, and hence, there may be multiple bench paths from $\mathcal{B}_h(\{s_I\})$ to goals. However, from a bench b, sequential GBFS explores only the *first* successor of b, according to the tie-breaking strategy, so only 1 bench path is explored. Due to this property, sequential GBFS in effect prunes large regions of some symmetric search spaces.

We extend and apply the notion of benches to analyze the behavior of KGBFS as an idealized model for HDGBFS. We first extend the notion of a bench. While GBFS explores bench $\mathcal{B}_h$, it will never expand a state s with $h(s) > hw_h(\mathcal{B}_h)$, because while exploring $\mathcal{B}_h$, there must be at least 1 state s' in OPEN such that $h(s') \leq hw_h(\mathcal{B}_h)$. On the other hand, KGBFS simultaneously expands k states, so it is possible that it expands a state s with $h(s) > hw_h(\mathcal{B}_h)$. Thus, we relax the definition of bench state candidate set in Definition 2 to include states s s.t. $h(s) > hw_h(\mathcal{B}_h)$.

When KGBFS expands $s \in \mathcal{B}_h$, and thereafter expands no more states in $\mathcal{B}_h$, we say that KGBFS exited $\mathcal{B}_h$ by expanding s. When KGBFS expands $s \in \mathcal{B}_h$ but does not exit $\mathcal{B}_h$, we say that KGBFS is searching $\mathcal{B}_h$.

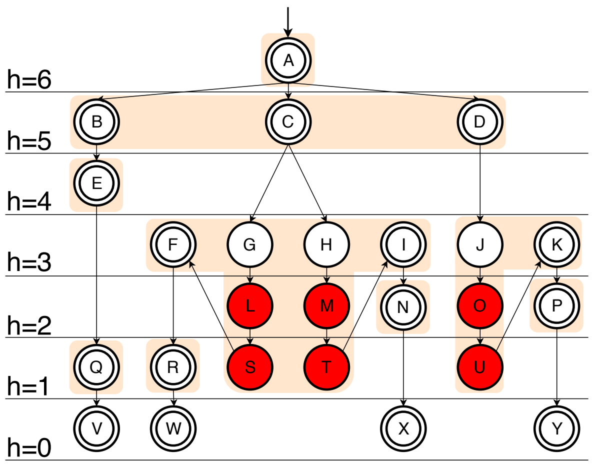


Figure 5: Example state space topology and bench transition system. States which belong to the same bench are surrounded by a color frame. The height of each state indicates its h-value. $s_I = A$, and $V, W, X, Y \in S_*$. Double circles indicate bench exit states or goal states.

Unlike GBFS, KGBFS expands multiple states simultaneously, so it is possible to search multiple benches simultaneously. The notion of a *crater* is proposed with respect to a bench, a set of states with a large high-water mark but low h-value, and it is known that when GBFS enters a crater, it must expand all states in the crater before expanding a bench exit state (Heusner, Keller, and Helmert 2017). When KGBFS searches multiple benches simultaneously, it is possible that it searches multiple craters before a bench exit state is expanded.

For example in Fig. 3.1, GBFS expands 5 states in the order $\langle A, B, E, Q, V \rangle$, so the bench path is $\langle \mathcal{B}_h(\{A\}), \mathcal{B}_h(\{B, C, D\}), \mathcal{B}_h(\{E\}), \mathcal{B}_h(\{Q\}) \rangle$ (assuming a tie-breaking policy where nodes on the left side of the figure have higher priority). In contrast, KGBFS with $k = 2$ expands 15 states in the order $\langle A, B, C, G, H, L, M, S, T, F, I, R, N, W \rangle$, and $\mathcal{B}_h(\{A\}), \mathcal{B}_h(\{B, C, D\}), \mathcal{B}_h(\{E\}), \mathcal{B}_h(\{G, H\}), \mathcal{B}_h(\{R\}), \mathcal{B}_h(\{N\})$. Thus, the search overhead of KGBFS$_{k=2}$ is 1.8, due to searching $\mathcal{B}_h(\{G, H\})$. KGBFS$_{k=2}$ must expand both craters in this bench and never expands E, although E is only two states away from a goal, V. With $k = 3$, KGBFS$_{k=3}$ expands 2 more benches, $\mathcal{B}_h(\{J\})$, which has 2 crater states, and $\mathcal{B}_h(\{P\})$ because KGBFS$_{k=3}$ expands 3 bench exit states B, C, D simultaneously. Thus, since KGBFS expand more states simultaneously as k increases, it sometimes expands multiple bench exit states and searches multiple benches simultaneously. Compared with GBFS, which searches only 1 bench path, KGBFS searches multiple bench paths, such as $\langle \mathcal{B}_h(\{G, H\}), \mathcal{B}_h(\{N\}) \rangle$ and $\langle \mathcal{B}_h(\{J\}), \mathcal{B}_h(\{P\}) \rangle$ in our example. This is often very wasteful, except in cases when KGBFS gets lucky and finds a shorter bench path than the one found by GBFS. In addition, if an expanded bench has many large craters with low h-value, such as $\mathcal{B}_h(\{G, H\})$ in our example, KGBFS must expand many states before exiting the bench. In other words, when some of the processes in HDGBFS find such "difficult benches", it is possible that due to hash-based distribution of states in such benches to all processes, all processes begin to search these benches, ignoring the "easy benches" which quickly lead to a solution. Since expanding many bench exit states increases the risk of searching "difficult benches", KGBFS with large k and HDGBFS with many processes are prone to incur large search overheads.

Note that searching multiple benches sometimes results in a reduction in expansions. Suppose that in Fig.3.1, we add an edge directly connecting B a goal V and GBFS and KGBFS search this search space with a tie-breaking policy where nodes on the right side of the figure have higher priority. While GBFS expands 8 states, $\langle A, D, J, O, U, K, P, Y\rangle$, KGBFS$_{k=3}$ expands 5 states, $\langle A, D, C, B, V\rangle$. Cases like this possibly account for the super-linear speedup of HDGBFS in Table 1.

Experimental Verification We experimentally analyzed the benches explored by these algorithms. Because of the resource requirements for state space analysis, we selected instances used for this experiment as follows: From the set of all instances from the IPC-11 and IPC-14 optimal tracks where GBFS expanded over 10 states, we chose 16 instances for which $\mathcal{S}$ could be constructed within 5 minutes using GBFS, and the number of states in $S \leq 10000$ (same instances listed in Table 2). This experiment uses the path-independent h_{ff} heuristic since the h_{lmc} heuristic value is path-dependent, making the analysis of $\mathcal{S}$ difficult.

For each instance, we first ran GBFS, but instead of stopping when a solution was found, we let GBFS run until the OPEN list was empty, recording all nodes, edges, and h-values in the state space $\mathcal{S}$ reachable by GBFS from the start state. From $\mathcal{S}$, we computed $\mathcal{B}_h(\{s_I\})$, as well as all benches induced by bench exit states s and $\mathcal{B}_h(succ(s))$, i.e., we computed the bench transition graph for $\mathcal{S}$.

Then, we ran GBFS and KGBFS (for k=4, 16), and HDGBFS (4, 16 processes) on these instances. We used $\mathcal{S}$ and the bench transition graph to identify the bench associated with each state visited. Since HDGBFS behaves nondeterministically due to its distributed nature, multi-process implementation, we took the average of 3 runs in order to stabilize the measurements.

The results are shown in Fig. 3.1. For KGBFS, on many instances, as k increases, both the number of expansions as well as the number of benches explored increases, consistent with our analysis. HDGBFS also displays the same tendency – as the number of processes increases, both the number of benches explored as well as nodes expanded tends to increase. These results suggest that the severe performance degradation of HDGBFS relative to GBFS that we saw in Sec. 2 can be explained by the fundamental difference in bench exploration behavior of GBFS vs. HDGBFS. Whereas GBFS is guaranteed to explore 1 bench at a time, making progress as it transitions from one bench to another, HDGBFS explores many benches simultaneously, resulting in a significant amount of search overhead.

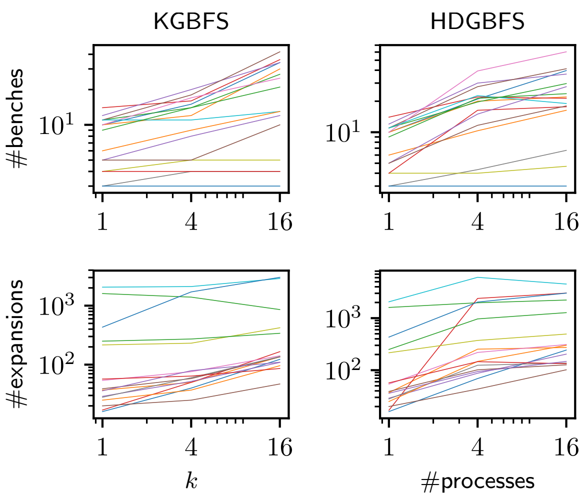


Figure 6: The number of expanded benches and nodes in KGBFS and HDGBFS (#processes=1 is GBFS). Each line represents 1 instance (color=domain).

4 Improved, Distributed GBFS

So far, we have shown that hash-distributed parallelization of GBFS can result in massive search overhead and slowdown compared to GBFS. Our analysis in the previous section showed that HDGBFS often fails to focus sufficiently greedily on promising areas (c.f., the maintenance example above), and ends up searching a much broader subset of the search space (i.e., more benches) than GBFS.

We now propose LG (Locally Greedy HDGBFS), which attempts to directly address this issue. In LG (Algorithm 1), while OPEN and CLOSED are used similarly to the OPEN and CLOSED lists in basic HDGBFS, each process has s^l, a "locally greedy state" which is used in order to force each processor to locally explore highly promising paths. s^l has a higher priority than OPEN – s^l is expanded before the best node in OPEN if not NULL. As with LE (Sec. 2.1), LG evaluates nodes at the processor where they are generated. When a processor generates a node s with h-value lower than the lowest h-value of s^l_i, s is assigned to s^l_i. This causes nodes which would be immediately expanded next by sequential GBFS (highest priority according to h and tie-breaking policy) to be expanded next instead of sent to its hash-assigned owner processor, and causes LG to greedily explore the path locally without being distracted by search regions received from other processors.

4.1 Experimental Evaluation

We compared LG with HDGBFS and GBFS, using the same experimental setup as in Sec. 2. Fig. 1 compares parallel GBFS vs. GBFS with respect to search time and the number of nodes expanded. Each colored point represents 1 instance (blue=HDGBFS, tan=LG), and the x-axis represents search time and #expansions for GBFS, allowing a 3-way comparison of LG vs HDGBFS vs GBFS. Instances solved by either GBFS or LG but not solved by the other are plotted

Algorithm 1 LG

```
1: for $i \leftarrow 1, ..., n$ do
2:     $h_i^{best} \leftarrow \infty$; $s_i^l \leftarrow$ NULL; $OPEN_i, CLOSED_i \leftarrow \emptyset$;
3: $s^l_{H(s_I) \bmod n} \leftarrow \{s_I\}$; $h^{best}_{H(s_I) \bmod n} \leftarrow h(s_I)$
4: In parallel, on each process $i$ execute 5-23.
5: loop
6:     $OPEN_i \leftarrow OPEN_i \cup \{s | s \in$ Received nodes $\setminus CLOSED_i\}$
7:     if $s_i^l \neq$ NULL or $OPEN_i \neq \emptyset$ then
8:         if $s_i^l \neq$ NULL then
9:             $s \leftarrow s_i^l$; $s_i^l \leftarrow$ NULL
10:        else
11:            $s \leftarrow \mathrm{argmin}_{s \in OPEN_i} h(s)$; $OPEN_i \leftarrow OPEN_i \setminus \{s\}$
12:        if $s \in S_*$ then return solution path
13:        if $s \notin CLOSED_i$ then
14:            $CLOSED_i \leftarrow CLOSED_i \cup \{s\}$
15:            $c_{best} \leftarrow \mathrm{argmin}_{c \in succ(s) \setminus CLOSED_i} h(c)$
16:            for $c \in succ(s) \setminus CLOSED_i$ do
17:                if $c = c_{best} \wedge h(c) < h_i^{best}$ then
18:                    $s_i^l \leftarrow c$; $h_i^{best} \leftarrow h(c)$;
19:                else if $i = H(c) \bmod n$ then
20:                    $OPEN_i \leftarrow OPEN_i \cup \{c\}$
21:                else
22:                    Add $c$ to the send buffer to $H(c) \bmod n$
23:            Send nodes
```

	GBFS		HDGBFS		LG	
	#e.	#b.	#e.	#b.	#e.	#b.
nomystery-p01	16.00	10.00	243.33	39.67	**151.00**	**28.33**
nomystery-p11	37.00	10.00	273.00	22.00	**224.00**	**21.67**
nomystery-p12	1604.00	11.00	2224.67	29.67	**1774.33**	**27.67**
nomystery-p13	17.00	14.00	3028.00	**21.33**	**299.33**	30.67
parcprinter-p01	29.00	12.00	203.33	36.67	**149.00**	**34.00**
parcprinter-p02	39.00	11.00	**126.67**	41.33	148.00	**35.00**
parcprinter-p03	54.00	10.00	310.67	60.33	**178.67**	**43.67**
sokoban-p01	216.00	4.00	**494.00**	**4.67**	530.67	5.00
sokoban-p03	2059.00	11.00	4518.00	**19.00**	**3892.67**	21.00
sokoban-p12	432.00	**3.00**	**3024.00**	**3.00**	3053.67	**3.00**
hiking-ptesting-1-2-3	25.00	6.00	301.00	**16.33**	**251.00**	16.67
hiking-ptesting-1-2-4	250.00	9.00	1274.67	23.67	**621.67**	**18.00**
maintenance-1-3-010-010-2-001	57.00	4.00	133.33	17.67	**95.67**	**11.33**
maintenance-1-3-010-010-2-002	36.00	5.00	147.33	27.67	**83.67**	**17.67**
tetris-p02-4	20.00	5.00	101.67	**18.00**	**87.67**	22.33

Table 2: # of expanded nodes (#e.) and benches explored (#b.) by GBFS/h_{ff}, HDGBFS/h_{ff}, and LG/h_{ff} on a set of IPC optimal-track instances. This experiment uses the path-independent h_{ff} heuristic since the h_{lmc} heuristic value is path-dependent.

as "unsolved". Table 1 shows per-domain results for LG vs. HDGBFS and LE.

In contrast to HDGBFS, for search time almost all of the points for LG are below the $1/1$ line, and for #expansions, the LG points tend to be much closer to the $1\times$ line than the HDGBFS points, indicating that LG successfully significantly reduces the search overhead compared to HDGBFS. Overall, LG coverage on the test instances is 287 (vs. 268 for HDGBFS and 272 for GBFS).

We re-emphasize that sequential GBFS uses 128GB RAM with a single core, while HDGBFS and LG use 8GB/core (total 128GB), so it is not surprising that with a sufficiently long time limit, sequential GBFS has a higher coverage than the parallel variants, as there is a tradeoff between RAM/core and total expansion rate across cores when using a single multicore machine. On the other hand, there are some instances such as in the thoughtful domain which are

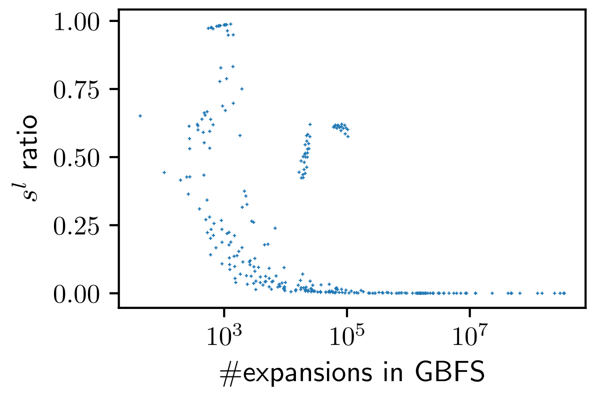


Figure 7: #expansions in GBFS/h_{lmc} vs. the fraction of expansions from s^l in LG/h_{lmc}.

solved very quickly by GBFS but not solved by LG which can not be explained by RAM fragmentation – improvements on such instances is future work.

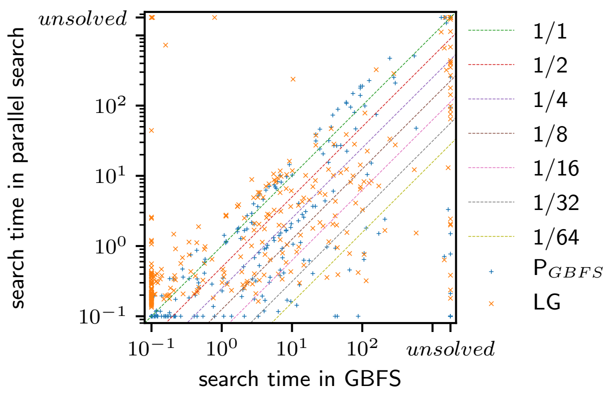
To compare the search behavior of LG vs. HDGBFS, Table 2 shows the number of benches explored by HDGBFS and LG (on average of 3 runs). LG tends to search fewer benches than HDGBFS, suggesting that these two parallel variants cover the search spaces in a different manner, and that overall, the LG behaves more similarly to GBFS than HDGBFS.

Next, we measure how often LG expands "locally greedy" nodes from its s^l, essentially behaving (locally) like GBFS. In Fig. 7, the x-axis represents the #expansions by GBFS, and the y-axis is the fraction of expansions from s^l. Each point represents a problem instance. When #expansions in GBFS is small, LG tends to have high y-values, and when #expansions in GBFS is large, the y-values for LG are very close to 0. Thus, s^l is used most frequently in instances which are solved quickly by GBFS. Recall from Sec. 2 that HDGBFS often incurs massive search overhead in such "GBFS-easy" problems, sometimes failing to solve the instances. This shows that for such problems, LG significantly improves upon HDGBFS by using s^l to aggressively follow a greedy path to a better h-value state on the local processor.

4.2 Comparison with a Parallel Portfolio

We have shown that LG succeeds in regaining some of the greedy behavior of GBFS and significantly outperforms HDGBFS. An alternative approach to utilizing many processors for search is a parallel portfolio, in which each processor independently performs a greedy search. P$_{GBFS}$ is a parallel portfolio which executes GBFS with the default (FIFO) tie-breaking on processor 1, LIFO tie-breaking on processor 2, and randomized tie-breaking with different random seeds on each of the other processors.

Fig. 8 compares search time for P$_{GBFS}$ on an Amazon EC2 r4.16xlarge instance (64 cores, 488 GiB RAM) and LG 64-cores on Amazon EC2 r4.xlarge instances (4 cores, 30.5GiB RAM $\times$ 16 machines). 16-core results are quali-



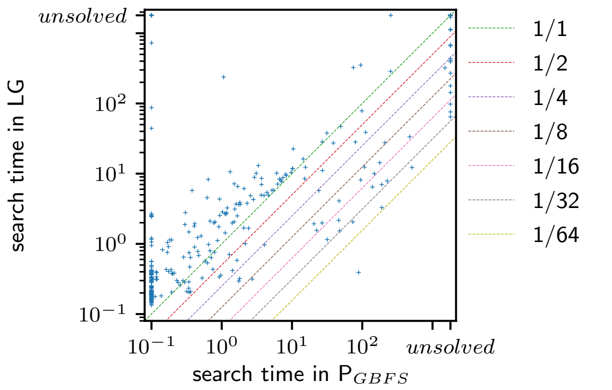


Figure 8: Search time: GBFS/h_{lmc} vs. P_{GBFS}/h_{lmc} (G) vs. LG/h_{lmc} (LG) on 64 processors (4 cores, 30.5GiB RAM $\times$ 16 machines,), 30min. The baseline GBFS in the top figure used a large machine with 488GiB RAM.

tatively similar (omitted due to space). The top figure compares runtime vs. runtime by a baseline single-process GBFS on the same machine as P_{GBFS}. Many of the P_{GBFS} points are close to the $1/1$ line – these correspond to the instances which are solved by the portfolio component using FIFO tie-breaking.

The bottom figure compares P_{GBFS} and LG runtimes. Overall, LG is competitive with P_{GBFS}, and they are complementary – LG performed well on instances where P_{GBFS} performed poorly, and vice versa.

Note that P_{GBFS} fails to solve 3 instances due to RAM exhaustion[1]. In several instances, LG exhausted RAM with 16 cores (64GB total), but successfully solved the instance with 64 cores (488GiB total), indicating that LG can successfully exploit the aggregate RAM of the system.

We have evaluated portfolio variants which attempt to partition the search near the root node as well as those which perform hash-based duplicate detection, but none have achieved better speedups than P_{GBFS}.

This shows that HDGBFS-based approaches are complementary to a parallel greedy portfolio approach. Thus, a promising direction for future work may be large-scale portfolios which allocate some processors to HDGBFS-based distributed parallel approaches such as LG, and other processors to independent greedy searches as in P_{GBFS}. Comparison with portfolios that exploit multiple heuristics and multiple search strategies, as well as the use of HDGBFS variants as components of large-scale portfolios are directions for future work.

[1]These instances were solved by the baseline single-process (488GiB of RAM) GBFS with FIFO tie-breaking, but P_{GBFS} (which includes a GBFS+FIFO tie-breaking component) fails because each process in the portfolio is only allocated 7.625 GiB so the FIFO component exhausts RAM before solving the instance.

5 Conclusion

This paper showed that hash-based work distribution, which has been shown to be quite effective for parallelizing A* because of its simplicity, load balancing, and duplicate detection, can result in disastrous performance degradation when straightforwardly applied to GBFS, a standard algorithm for satisficing search.

We investigated the search overhead of HDGBFS and showed that unlike GBFS, which explores a sequence of benches (Heusner, Keller, and Helmert 2017), HDGBFS explores many benches simultaneously. Unlike HDA*, which basically behaves similarly to A*, HDGBFS behaves very differently from GBFS, so unlike parallelization of A*, where the main issues were efficient duplicate detection and reduction of communication and synchronization overheads, our analysis shows that successful parallelization of GBFS poses a nontrivial algorithmic challenge.

We proposed LG, which seeks to address the search overhead of HDGBFS by forcing each processor to pursue highly promising local paths locally rather than sending all nodes to their hash-based owner process. We showed that LG significantly outperforms HDGBFS, and is competitive with and complementary to a parallel portfolio consisting of independent GBFS runs with different tie-breaking policies, so LG represents a promising step towards efficient, distributed, satisficing search. However, there is no theoretical guarantee of performance degradation of LG, and LG actually fails to solve some instances solved by GBFS. Investigating domains where LG causes the pathological behavior and improving parallel GBFS on those domains are future work.

A significant motivation for parallelizing search is to exploit large amounts of aggregate RAM in a cluster with many machines in order to solve problems which cannot be solved using one machine, so an effective, distributed memory parallelization is necessary. We showed that LG can scale fairly well with up to 64 cores (4 cores $\times$ 16 machines). Future work will investigate further scaling of HDGBFS-based approaches on larger systems. In a single machine, multi-core environment, other approaches such as PBNF can be used to parallelize satisficing search (Burns et al. 2010).

GBFS is an instance of Weighted A* (WA*) (Pohl 1970), where $w_h = 1, w_g = 0$. Although HDA* has been shown to be quite effective for admissible search using A* (WA* with $w_h = w_g$), preliminary results indicate that as w_h is increased, HDA* will incur large search overhead, similar to HDGBFS. An investigation of distributed WA* is a direction for future work.

References

Asai, M., and Fukunaga, A. 2017. Exploration among and within plateaus in greedy best-first search. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017.*, 11–19.

Burns, E.; Lemons, S.; Ruml, W.; and Zhou, R. 2010. Best-first heuristic search for multicore machines. *J. Artif. Intell. Res.* 39:689–743.

Doran, J., and Michie, D. 1966. Experiments with the graph traverser program. In *Proceedings of The Royal Society A: Mathematical, Physical and Engineering Sciences*, volume 294, 235–259.

Felner, A.; Kraus, S.; and Korf, R. E. 2003. KBFS: k-best-first search. *Ann. Math. Artif. Intell.* 39(1-2):19–39.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on* 4(2):100–107.

Heusner, M.; Keller, T.; and Helmert, M. 2017. Understanding the search behaviour of greedy best-first search. In *Proceedings of the Tenth International Symposium on Combinatorial Search, SOCS 2017, 16-17 June 2017, Pittsburgh, Pennsylvania, USA.*, 47–55.

Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation through Heuristic Search. *J. Artif. Intell. Res* 14:253–302.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *J. Artif. Intell. Res.* 22:215–278.

Jinnai, Y., and Fukunaga, A. 2017. On hash-based work distribution methods for parallel best-first search. *J. Artif. Intell. Res.* 60:491–548.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2013. Evaluation of a simple, scalable, parallel best-first search strategy. *Artif. Intell.* 195:222–248.

Pohl, I. 1970. Heuristic search viewed as path finding in a graph. *Artif. Intell.* 1:193–204.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res.* 39:127–177.

Vidal, V.; Bordeaux, L.; and Hamadi, Y. 2010. Adaptive k-parallel best-first search: A simple but efficient algorithm for multi-core domain-independent planning. In *Proceedings of the Third Annual Symposium on Combinatorial Search, SOCS 2010, Stone Mountain, Atlanta, Georgia, USA, July 8-10, 2010*, 100–107.

Wilt, C. M., and Ruml, W. 2014. Speedy versus greedy search. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014, Prague, Czech Republic, 15-17 August 2014.*, 184–192.

Xie, F.; Müller, M.; and Holte, R. 2014. Adding local exploration to greedy best-first search in satisficing planning. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, 2388–2394.

Zobrist, A. L. 1970. A new hashing method with applications for game playing. Technical report, Dept of CS, Univ. of Wisconsin, Madison. Reprinted in *International Computer Chess Association Journal*, 13(2):169-173, 1990.