GONDOR to the Rescue: Satisficing Planning with Low Memory

Yonatan Feanor Vernik[1], **Alexander Tuisov**[2], **Alexander Shleyfman**[1]

[1]Computer Science Department, Bar-Ilan University
[2]Independent Researcher
{yonatanw55, queldelan}@gmail.com, alexander.shleyfman@biu.ac.il

Abstract

Greedy Best-First Search (GBFS) is the dominant approach for solving search problems where the goal can be estimated with a heuristic, such as planning, route finding, navigation, and pathfinding. This is especially true when the memory is tightly constrained, such as planning on edge devices. To alleviate that, we present GONDOR (Greedy Online Navigation with Dynamic Outpost-based Re-search), a memory-efficient extension of GBFS that allows search to continue under strict memory limits by periodically compressing the search tree while retaining a sparse set of anchor states, then upon reaching the goal reconstructs the path by re-searching between the sparse states. We analyze the algorithm and discuss several variants defined by different outpost selection policies. In addition, we explore using Bloom filters for compact duplicate detection in the closed list. Experiments across numeric planning domains and heuristic configurations show that GONDOR consistently improves coverage under low memory budgets compared to standard GBFS. We release the implementation of GONDOR and the Bloom-filter variant to facilitate further research on memory-efficient heuristic search.

Introduction

In the past two decades Greedy Best-First Search (GBFS) de facto became the state-of-the-art for obtaining fast solutions for satisficing planning, mostly due to strong empirical performance when guided by informative heuristics (Bonet and Geffner 2001; Helmert 2006; Taitler et al. 2024). In this setting, GBFS maintains an open list ordered by a heuristics h and repeatedly expands the node with the smallest value, generating successors that are inserted back into the open list while expanded nodes are moved to a closed list.

This greedy selection strategy often leads to rapid progress toward a solution, but it also introduces significant memory demands because the algorithm stores all nodes. Since GBFS does not use path-cost information to prune the search, large plateaus of nodes with similar heuristic values may accumulate, causing the search frontier to grow substantially (Imai and Kishimoto 2011).

A further complication is that GBFS is highly sensitive to heuristic inaccuracies. If the heuristic misguides the search, the algorithm may devote extensive effort to regions of the state space that do not contribute to a solution, expanding many nodes that are ultimately irrelevant. This phenomenon has been documented in studies analyzing GBFS search behavior, which show that the algorithm may expand large sets of states depending on tie-breaking and heuristic structure, even when those states do not lie on promising paths (Heusner, Keller, and Helmert 2017, 2018a).

Because both expanded nodes and generated-but-unexpanded nodes must be retained in memory, this behavior can lead to substantial memory consumption. Thus, GBFS may therefore exhaust available memory even after making substantial progress toward a goal, particularly when the heuristic induces broad plateaus or misleading gradients that cause extensive exploration before a solution is reached (Heusner, Keller, and Helmert 2018b; Kuroiwa and Beck 2022). This limitation motivates many extensions to GBFS, including diversification strategies (Imai and Kishimoto 2011) and multi-queue variants (Richter and Westphal 2010; Xie, Müller, and Holte 2015).

To alleviate this, we introduce GONDOR (Greedy Online Navigation with Dynamic Outpost-based Re-search), that adds a memory-bounding mechanism to GBFS. When memory limits are reached, the algorithm prunes the search tree, retaining only a sparse subset of states called outposts. GONDOR then resumes the search from these outposts, effectively managing memory while maintaining progress.

Once a goal is reached, the outposts along the discovered route define a sequence of beacons. In a second phase, a complete plan is reconstructed by solving smaller exact-state subproblems between consecutive beacons. We discuss outpost-selection policies and perform experiments with a naive Bernoulli policy, as well as with a Bloom Filter-based closed list to further reduce memory consumption, tested on both standard memory and low-memory settings.

Much of the work on memory-efficient heuristic search has focused on optimal planning. In contrast, we consider standard fully observable deterministic satisficing planning (Ghallab, Nau, and Traverso 2004). The MA* algorithm (Chakrabarti et al. 1989; Russell 1992) also frees memory by removing unpromising states, but does so incrementally by "de-growing" the search tree one node at a time. Frontier search (Korf et al. 2005) discards entire portions of the search tree, similar in spirit to our approach; however it does

so incrementally and reconstructs the solution using bidirectional search, whereas we we drop at once on memory pressure, and reconstruct using short re-searches with beacons.

Background

We consider search algorithms defined over a **state space** $\mathcal{S} = \langle S, s_I, S_\star, \mathrm{succ} \rangle$, where S is a non-necessarily finite set of states, $s_I \in S$ is the initial state, $S_\star \subseteq S$ is the set of goal states, and $\mathrm{succ} : S \to 2^S$ is a successor function that maps each state $s \in S$ to the finite set of its successor states.

A sequence of pairwise distinct states $\langle s_0, s_1, \ldots, s_n \rangle$ is a path from s_0 to s_n if $s_i \in \mathrm{succ}(s_{i-1})$ for $i \in \{1, \ldots, n\}$. A path $\langle s_0, \ldots, s_n \rangle$ is a s-plan if $s_0 = s$ and $s_n \in S_\star$.

Heuristic $h : S \to \mathbb{R}_{\geq 0}$ is a function. Typically, $h(s)$ estimates the cost of a cheapest s-plan, but this is not a necessary condition in the satisficing setting. In this work, the heuristic is treated as an arbitrary black-box function that assigns a finite non-negative real value to each state.

Greedy Best-First Search (GBFS) takes as input a state-space $\mathcal{S}$ and returns an s_I-plan if one exists, and *unsolvable* otherwise. GBFS is guided by the assumption that states with lower heuristic values are closer to a goal state. At each iteration, the algorithm expands a generated state with minimum heuristic value (taken from the OPEN list), and terminates once a goal state is generated. To reduce memory consumption, all generated states are kept in the CLOSED list, and no state is expanded twice. Due to greediness, GBFS provides no guarantee on the quality of the resulting s_I-plan.

Bloom filter is a compact probabilistic data structure with a constant memory use for approximate set membership. It supports insertions and membership queries, may yield false positives, and does not yield false negatives. For further info see (Bloom 1970; Bonomi et al. 2006). In some of our experiments below we use Bloom filter to represent the closed list, reducing memory at the cost of occasionally pruning a previously unseen state.

Search with Limited Memory

We introduce two intermediate observations that motivate the design of GONDOR. These methods illustrate different ways of adapting GBFS to operate under strict memory limits. Consider the following algorithms.

GBFS-U (Unlimited) is a variant of GBFS that does not terminate when the memory bound is exceeded. Instead, after each node expansion, if memory usage surpasses a predefined threshold, the algorithm repeatedly removes the oldest node from the search tree (and associated data structures) until the memory usage falls below the threshold. The search then continues from the remaining frontier.

Because nodes may be removed during search, parent pointers required for solution reconstruction may be lost. Thus, GBFS-U may only verify the existence of a solution, but it does not necessarily preserve a complete plan.

GBFS-R (Retrace) extends GBFS-U by exploiting the observation that when a goal state is reached, a suffix of the plan remains intact from a surviving ancestor to the goal. The algorithm iteratively reconstructs the full plan by repeatedly running GBFS-U: each run searches from the initial state to the earliest surviving ancestor of the previously recovered suffix. The recovered suffixes are concatenated until a complete plan is obtained.

This approach preserves the reduced memory footprint of GBFS-U while enabling plan reconstruction. However, it may require multiple re-searches from the initial state, which can substantially increase runtime.

GONDOR Algorithm 1 modifies the retracing procedure by replacing repeated searches from the initial state with searches over shorter, non-overlapping segments. During the initial search, a subset of generated nodes is designated as *outposts*. Each node additionally stores a pointer to its nearest ancestral outpost; the initial state is always an outpost.

Search proceeds similarly to GBFS until memory usage exceeds a threshold L at which point a cleanup procedure REMOVENONOUTPOSTS is triggered. During cleanup, all non-outpost nodes are discarded (this includes the search tree, the OPEN list, and the and the CLOSED list). The search then resumes from the remaining outposts. Ordinary parent pointers of outposts are reset, while outpost-parent pointers are preserved. This effectively compresses the explored search history into a sparse chain of anchoring states.

When a goal is found, GONDOR first extracts the remaining suffix of the plan using standard parent pointers.[1] The algorithm then follows outpost-parent pointers to obtain an ordered sequence of outposts along the discovered route, referred to as *beacons*. The search tree is subsequently cleared, and a second phase reconstructs the full plan by re-solving the subproblems between consecutive beacons and concatenating the resulting segments with the retained suffix.

GONDOR includes a function hyper-parameter *IsOutpost*, which determines whether a generated node is marked as an outpost. Here, we use a simple Bernoulli policy in which each node independently becomes an outpost with probability p. Smaller values of p increase the effective memory available to the search, but may increase the cost of re-expansion after cleanup and the time required for plan reconstruction. Empirically, the reconstruction time for $p = 0.01$ was typically small relative to the initial search.

Theoretical Properties Note that GBFS is complete on finite search spaces and does not have such guarantees on infinite ones. Since no algorithm can solve a plan existence problem within constant memory, we analyze GONDOR under a setting where the memory available for each iteration is bounded by a constant L, where L is measured in the number of newly generated nodes.

Consider the stochastic outpost policy ISOUTPOST $\sim \mathrm{Ber}(p)$. When $p = 0$ and there is no threshold L, the algorithm reduces to the execution of GBFS. In this setting, assuming the same h and tie breaking the algorithm returns the same solution as GBFS.

For $p > 0$, additional memory is allocated to storing outposts, reducing the effective memory available to the main search. However, after each cleanup, the search restarts from a frontier containing the initial state and a subset of previously explored states, preserving soundness.

[1] This suffix always terminates at an outpost, since after each cleanup only outposts remain in the OPEN list.

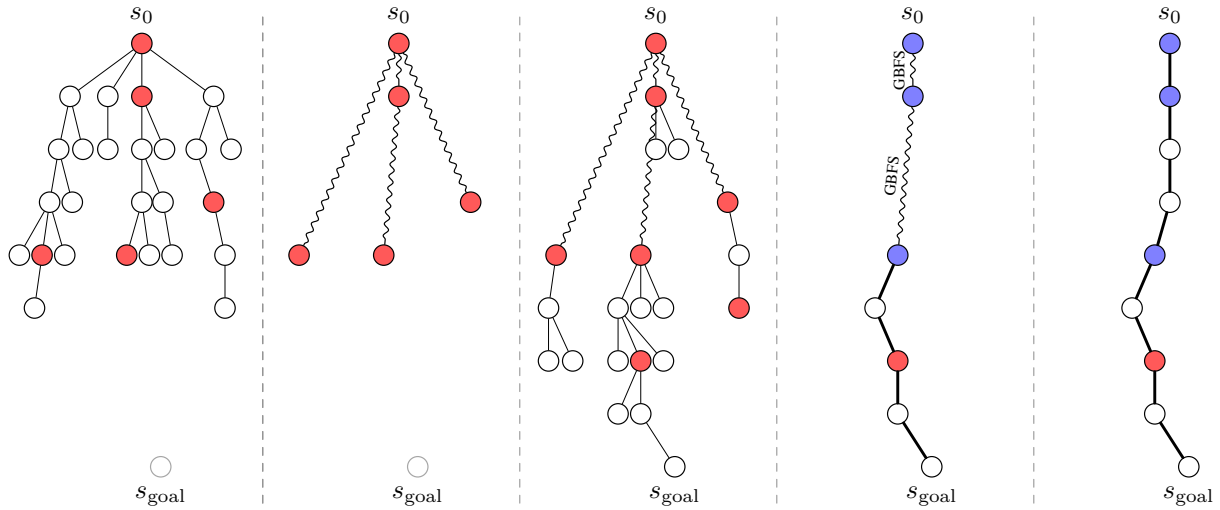

Figure 1: Illustration of GONDOR. Our method starts by searching with GBFS expansion order (left). Some search nodes, always including s_0, are designated as outposts (in red). Once memory runs out, all the nodes are cleared except for the outposts, connected by $outpost\_parent$ pointers (left middle). Search continues from there (middle) until a goal is reached, with repeated clears every time the memory limit is reached. Once a goal is reached, we trace along the path until $parent = \perp$, then trace along $outpost\_parent$ to the start (right middle). Outposts along the missing segments are promoted to beacons (blue) and the rest of the tree is dropped. Finally, the missing path segments are determined by re-searching with GBFS and the results are concatenated, along with the plan suffix before, to create the final path (right).

If GBFS with unbounded memory follows a solution path, then GONDOR expands progressively longer prefixes of that path. Since each generated node is selected as an outpost with probability $p > 0$, the probability that none of the prefix states is retained decreases exponentially with its length. Thus, progress to the goal is preserved across cleanups. When later states have lower heuristic values, cleanup primarily removes less promising regions.

After a goal is found, only the beacons and final suffix are kept. Reconstruction solves beacon-to-beacon segments in reverse order; each lies in a region already explored with at least the same memory, so it expands no more states than the original search. Thus, under the same heuristic and tie-breaking, reconstruction causes no additional memory overflows.

Proposition 1. *Given a bound L on the number of newly generated nodes per cleanup phase and $p > 0$,* GONDOR *is almost surely complete on finite state spaces.*

Proof. The algorithm terminates when a goal is found or when the OPEN list is empty. Since the initial state is always an outpost, an empty OPEN list implies that all reachable states were explored and the instance is unsolvable.

Between two cleanup phases, at most L nodes are generated, and each is marked as an outpost independently with probability $p > 0$. Hence, the probability of adding at least one new outpost in a phase is strictly positive. If no outpost is added, the same phase repeats. Therefore, with probability 1, after finitely many phases at least one additional state is added to the outpost set.

Because the state space is finite, eventually either a goal is found or all reachable states become outposts. In the latter case, no nodes are deleted and the remaining search exhausts the OPEN list, reporting failure. Thus, GONDOR is almost surely complete. □

Experimental results

The experiments were conducted on a 13th Gen Intel® Core™ i9-13900 processor running at 2.00 GHz, supported by a 64-bit operating system and 32 GB of RAM. To evaluate our search algorithm on a diverse set of heuristics and domains, we follow Tuisov, Vernik, and Shleyfman (2026) to generate 10 domain-specific heuristics[2] using GPT-5.1 with high reasoning effort and the default *temperature*=1.0 and *top_p*=1.0. We have evaluated our approach on the domains from the Numeric International Planning Competition (IPC) 2023 (Taitler et al. 2024) as well as on the two domains introduced in Tuisov, Vernik, and Shleyfman (2026); TwinPrimes and Pacman.

We ran each heuristic on each instance with a time limit of 10 minutes and a RAM limit of 8 GB (or 512 MB for low-memory experiments). We searched with the following settings: GBFS, GBFS$_{BF}$, GONDOR, and GONDOR$_{BF}$. All GONDOR settings use $p = 0.01$ as described above, while the Bloom Filter settings used k=23 hash functions and 400MiB of RAM (fpr=10^{-7} for set membership check at 10^8 nodes) for 8 GB limit, and k=20 and 3.5MiB RAM (fpr=10^{-6} for set membership check at 10^6 nodes) for 512 MB limit setting (Bonomi et al. 2006).

[2]The input to the LLM includes the domain and transition rules, but not the goal or the initial state which are different per instance

Algorithm 1: GONDOR

Input: initial state s_0, goal test S_*, heuristic h, successor function SUCC, memory limit L
Input: outpost policy ISOUTPOST
Output: plan π or failure
$r \leftarrow \text{NEWNODE}(s_0, \bot)$
$r.po \leftarrow \bot$; $r.is\_outpost \leftarrow True$ // po is parent_outpost
OPEN $\leftarrow \{r\}$; CLOSED $\leftarrow \{s_0\}$
while OPEN $\neq \emptyset$ **do**
    $n \leftarrow$ pop node from OPEN according to h
    if $n.state \in S_*$ **then**
        return RECONSTRUCTVIABEACONS(n, h)
    end if
    for each $s' \in \text{SUCC}(n.state)$ **do**
        if $s' \in$ CLOSED **then**
            continue
        end if
        $m \leftarrow \text{NEWNODE}(s', n)$
        $m.po \leftarrow n$ **if** $n.is\_outpost$ **else** $n.po$
        $m.is\_outpost \leftarrow \text{ISOUTPOST}(m)$
        OPEN $\leftarrow$ OPEN $\cup \{m\}$
        CLOSED $\leftarrow$ CLOSED $\cup \{m.state\}$
    end for
    if memory usage exceeds L **then**
        OPEN $\leftarrow$ REMOVENONOUTPOSTS(OPEN)
        CLOSED $\leftarrow$ OPEN.$state$
    end if
end while
return failure

Table 1 shows the coverage results. On the best heuristics all suggested methods equal or beat the baseline on all domains in both memory settings. For 8GB, gains are primarily in Counters (+4) and Drone (+1), while for 512 MB gains are more spread out including (+7) on TPP. When considering all heuristics, the suggested methods no longer dominate per-domain and every method has some domain it is best on, but GONDOR$_{BF}$ wins on most of the domains and overall.

As shown in Figure 2, GONDOR is slightly slower, though the gap decreases with increasing problem size. While GBFS frequently exhausts available memory after roughly 30–200 seconds (10–40 under tighter limits), GONDOR continues beyond this point and often succeeds. Memory cleanups performed by GONDOR also sometimes reduce runtime by discarding unpromising regions. The tradeoff is that when GONDOR fails, it typically consumes a larger portion of the time budget, whereas GBFS often fails earlier. For detailed results see Appendix.

Conclusion and future work

We present a new algorithm, GONDOR, which extends GBFS for memory efficient satisficing search, and evaluate a simple variation on numeric planning problems. The experimental results show that even with our naive hyperparameter selection GONDOR yields consistent benefits to coverage which is further improved by using a Bloom Filter as CLOSED list, and the benefit is $\sim 3\times$ larger for low-memory settings, at the cost of *failing slower* when it does fail.

Future work should explore more informed *IsOutpost* functions. One promising option is to increase outpost probability p with the relative heuristic improvement $h(s_0) - h(s)$ over the best state reached so far. Another is to assign outpost status during cleanup rather than at node generation.

	Metric	GBFS	GBFS$_{BF}$	GONDOR	GONDOR$_{BF}$
8 GB	Max Coverage (400)	302	302	307	308
	Sum Coverage (4000)	2339	2377	2385	2406
512 MB	Max Coverage (400)	285	290	296	298
	Sum Coverage (4000)	2000	2060	2099	2196

Table 1: *Max* sums the coverage of the **best** heuristic in each domain, while *Sum* sums the coverage of **all** heuristics.

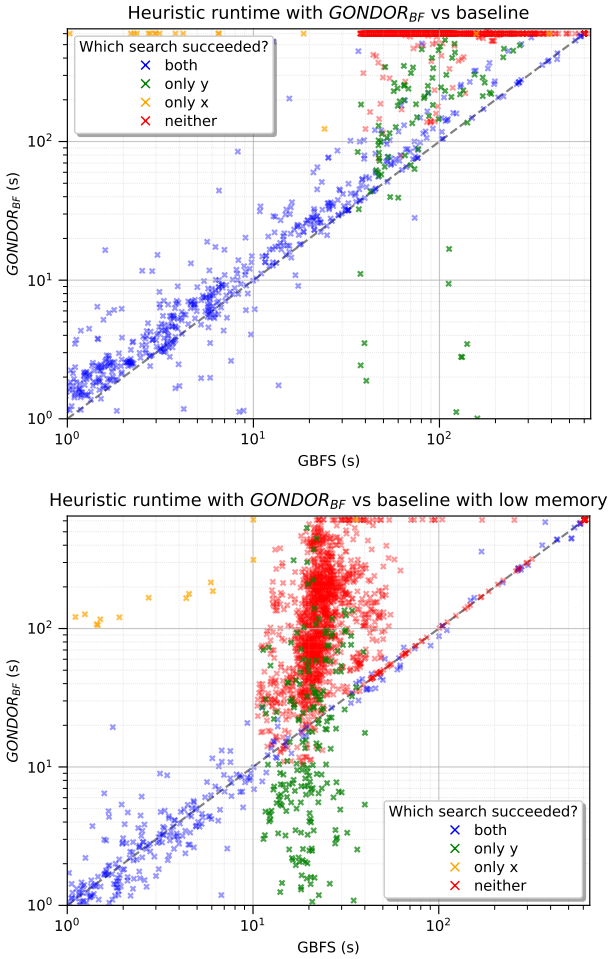

Figure 2: Runtime comparison for GONDOR$_{BF}$ and GBFS across heuristics and instances. Each point is one heuristic X instance run, colored by the solving method: green for GONDOR$_{BF}$, orange for GBFS. Points below the diagonal favor GONDOR$_{BF}$; points above favor GBFS. Runs under 1s are omitted for clarity.

Acknowledgments

This work was partially supported by ISF grant 2443/23.

References

Bloom, B. H. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7): 422–426.

Bonet, B.; and Geffner, H. 2001. Planning as Heuristic Search. *AIJ*, 129(1): 5–33.

Bonomi, F.; Mitzenmacher, M.; Panigrahy, R.; Singh, S.; and Varghese, G. 2006. An Improved Construction for Counting Bloom Filters. In Azar, Y.; and Erlebach, T., eds., *Algorithms – ESA 2006*, 684–695. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-38876-0.

Chakrabarti, P. P.; Ghose, S.; Acharya, A.; and Sarkar, S. C. D. 1989. Heuristic Search in Restricted Memory. *Artif. Intell.*, 41(2): 197–221.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Helmert, M. 2006. The Fast Downward Planning System. *JAIR*, 26: 191–246.

Heusner, M.; Keller, T.; and Helmert, M. 2017. Understanding the Search Behaviour of Greedy Best-First Search. In *SOCS*, 47–55. AAAI Press.

Heusner, M.; Keller, T.; and Helmert, M. 2018a. Best-Case and Worst-Case Behavior of Greedy Best-First Search. In *IJCAI*, 1463–1470. ijcai.org.

Heusner, M.; Keller, T.; and Helmert, M. 2018b. Search Progress and Potentially Expanded States in Greedy Best-First Search. In *IJCAI*, 5269–5273. ijcai.org.

Imai, T.; and Kishimoto, A. 2011. A Novel Technique for Avoiding Plateaus of Greedy Best-First Search in Satisficing Planning. In *SOCS*, 197–198. AAAI Press.

Korf, R. E.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier Search. *Journal of the ACM*, 52(5): 715–748.

Kuroiwa, R.; and Beck, J. C. 2022. Biased Exploration for Satisficing Heuristic Search. In *ICAPS*, 213–221. AAAI Press.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *JAIR*, 39: 127–177.

Russell, S. J. 1992. Efficient Memory-Bounded Search Methods. In *ECAI 1992*, 1–5.

Taitler, A.; Alford, R.; Espasa, J.; Behnke, G.; Fiser, D.; Gimelfarb, M.; Pommerening, F.; Sanner, S.; Scala, E.; Schreiber, D.; Segovia-Aguas, J.; and Seipp, J. 2024. The 2023 International Planning Competition. *AI Mag.*, 45(2): 280–296.

Tuisov, A.; Vernik, Y.; and Shleyfman, A. 2026. Successor-Generator Planning with LLM-generated Heuristics. arXiv:2501.18784.

Xie, F.; Müller, M.; and Holte, R. 2015. Understanding and Improving Local Exploration for GBFS. In *ICAPS*, 244–248. AAAI Press.

Appendix for GONDOR to the Rescue: Satisficing Planning with Low Memory

Yonatan Feanor Vernik[1], **Alexander Tuisov**[2], **Alexander Shleyfman**[1]

[1]Computer Science Department, Bar-Ilan University
[2]Independent Researcher
{yonatanw55, queldelan}@gmail.com, alexander.shleyfman@biu.ac.il

algorithms

We reproduce here pseudocode for all algorithms we describe.

GBFS-unlimited

We reproduce here the GBFS-Unlimited algorithm.

Algorithm 1: GBFS-Unlimited

Input: initial state s_0, goal condition G, heuristic h, successor function SUCC, memory limit L
Output: $(status, n)$
create root node r for s_0
$r.parent \leftarrow \perp$; $r.action \leftarrow \perp$; $r.time \leftarrow 0$
$OPEN \leftarrow \{r\}$; $CLOSED \leftarrow \{r\}$; $t \leftarrow 1$
while $OPEN \neq \emptyset$ **do**
    $n \leftarrow$ pop best node from $OPEN$
    if $n.state \models G$ **then**
        return $(\text{SUCCESS}, n)$
    end if
    for each $(a, s') \in \text{SUCC}(n.state)$ **do**
        if $s' \in CLOSED$ **then**
            continue
        end if
        create child node m for s'
        $m.parent \leftarrow n$; $m.action \leftarrow a$; $m.time \leftarrow t$
        $t \leftarrow t + 1$
        insert m into $OPEN$
        insert m into $CLOSED$
    end for
    while memory usage exceeds L and $OPEN \neq \emptyset$ **do**
        remove from $OPEN$ and from $CLOSED$ the node with smallest $time$
    end while
end while
return $(\text{FAILURE}, \perp)$

GBFS-retrace

We reproduce here the GBFS-Retrace algorithm.

Algorithm 2: GBFS-Retrace

Input: initial state s_0, goal condition G, heuristic h, successor function SUCC, memory limit L
Output: plan π or failure
$\pi \leftarrow [\,]$
while $True$ **do**
    $(status, n) \leftarrow GBFS - Unlimited(s_0, G, h, SUCC, L)$
    if status is FAILURE **then**
        break
    end if
    while $n.parent \neq None$ **do**
        $push\ n.action\ into\ \pi$
        $n \leftarrow n.parent$
    end while
    if $n.state == s_0$ **then**
        ▷ we reverse plan since it was pushed end first
        return $REVERSE(\pi)$
    end if
    $G \leftarrow lambda\ s \in S : s == n$
end while
return FAILURE

GONDOR

The algorithm is already in the paper, but it's repeated here for proximity.

Algorithm 3: GONDOR

Input: initial state s_0, goal test S_*, heuristic h, successor function SUCC, memory limit L
Input: outpost policy ISOUTPOST
Output: plan π or failure
$r \leftarrow \text{NEWNODE}(s_0, \bot)$
$r.po \leftarrow \bot$; $r.is\_outpost \leftarrow True$ // po is parent_outpost
OPEN $\leftarrow \{r\}$; CLOSED $\leftarrow \{s_0\}$
while OPEN $\neq \emptyset$ **do**
    $n \leftarrow$ pop node from OPEN according to h
    if $n.state \in S_*$ **then**
        return RECONSTRUCTVIABEACONS(n, h)
    end if
    for each $s' \in \text{SUCC}(n.state)$ **do**
        if $s' \in$ CLOSED **then**
            continue
        end if
        $m \leftarrow \text{NEWNODE}(s', n)$
        $m.po \leftarrow n$ **if** $n.is\_outpost$ **else** $n.po$
        $m.is\_outpost \leftarrow \text{ISOUTPOST}(m)$
        OPEN $\leftarrow$ OPEN $\cup \{m\}$
        CLOSED $\leftarrow$ CLOSED $\cup \{m.state\}$
    end for
    if memory usage exceeds L **then**
        OPEN $\leftarrow$ REMOVENONOUTPOSTS(OPEN)
        CLOSED $\leftarrow$ OPEN$.state$
    end if
end while
return failure

ReconstructViaBeacons

We reproduce here the ReconstructViaBeacons algorithm.

Algorithm 4: RECONSTRUCTVIABEACONS

Input: goal node n, heuristic h
Output: complete plan π
$\pi_{suffix} \leftarrow [\,]$; $\mathcal{B} \leftarrow [\,]$ ▷ store the suffix since last clear
while $n.parent \neq \bot$ **do**
    push $n.action$ to π_{suffix}
    $n \leftarrow n.parent$
end while ▷ light the beacons
$\pi_{suffix} \leftarrow REVERSE(\pi_{suffix})$
while $n \neq \bot$ **do**
    push $n.state$ to $\mathcal{B}$
    $n \leftarrow n.parent\_outpost$
end while
▷ follow the beacons. NOTE: they were added in reverse order
for $i = 1$ to $|\mathcal{B}| - 1$ **do**
    $G \leftarrow lambda\ s \in S : s == \mathcal{B}[i]$
    $\pi_i \leftarrow \text{GBFS}(s_0 = \mathcal{B}[i+1], G, h)$
    prepend π_i to π_{suffix}
end for
return π_{suffix}

definitions

Definition 1 (Weak Landmark Set)**.** *Let $\mathcal{S} = \langle S, s_I, S^\star, \text{succ}\rangle$ be a state space. A set $W \subseteq S$ is a* weak landmark set *for $\mathcal{S}$ if there exists an s_I-plan $\pi = \langle s_0, s_1, \ldots, s_n\rangle$ such that $W \subseteq \{s_0, s_1, \ldots, s_n\}$.*

This contrasts with the classical notion of a strong landmark, which is an action or fact that must appear in *every* s_I-plan. A weak landmark set relaxes this requirement to existence: suffices that all its elements are collectively visited by at least one plan.

Definition 2 (Beacon Sequence)**.** *A* beacon sequence *for $\mathcal{S}$ is an ordered tuple $\mathcal{B} = (b_0, b_1, \ldots, b_k)$ of states satisfying:*

1. *$\{b_0, b_1, \ldots, b_k\}$ is a weak landmark set for $\mathcal{S}$, and*
2. *there exists an s_I-plan $\langle s_0, s_1, \ldots, s_n\rangle$ in which the beacons appear in order: there exist indices $0 \le i_0 < i_1 < \cdots < i_k \le n$ such that $s_{i_j} = b_j$ for all $j \in \{0, \ldots, k\}$.*

Intuitively, a beacon sequence is a weak landmark set whose order we know along that plan. Every weak landmark set has at least one Beacon sequence, which can be constructed by indexing the states along π. In GONDOR, phase 1 constructs a valid π, for which the outposts (or in fact, any subset of states indexed along it) are a valid beacon sequence.

Analogy to Weak and Strong Stubborn Sets. The weak/strong relation introduced above for landmark sets mirrors an analogous relation in the stubborn-set literature (Valmari 1991; Sievers and Wehrle 2021). Stubborn sets are a partial-order-reduction technique that prunes the successors considered at each search state. *Strong* stubborn sets (SSS) impose the requirement that every optimal plan in the original space corresponds to a plan of equal cost in the pruned space—a universal requirement over the entire solution set.

Weak stubborn sets (WSS), as defined by Valmari in the original formulation and recently revisited for classical planning by Sievers and Wehrle (2021), relax this to an existential guarantee: the pruned space is only required to preserve *at least one* optimal plan, permitting more aggressive pruning.

Strong landmarks and strong stubborn sets both satisfy a *universal* condition, whereas the weak landmark sets and weak stubborn sets both satisfy an *existential* condition. Every strong stubborn set is also a weak one, and every set of (strong) state landmarks is also a weak landmark set. GONDOR's beacons are not landmarks, but they are a weak landmark set which suffices for path reconstruction.

coverage results

Below are the per-domain coverage results

Domain	GBFS		GONDOR	
	no-BF	BF	no-BF	BF
Block Grouping (20)	19	19	19	19
Counters (20)	10	10	**14**	**14**
Delivery (20)	18	18	**19**	18
Drone (20)	19	19	19	**20**
Expedition (20)	4	4	4	4
Farming (20)	20	20	20	20
FO-Counters (20)	7	7	7	7
FO-Farming (20)	20	20	20	20
FO-Sailing (20)	20	20	20	20
Hydropower (20)	20	20	20	20
Market Trader (20)	20	20	20	20
Pacman (20)	16	16	16	16
Pathways (20)	1	1	1	2
Plant Watering (20)	20	20	20	20
Rover (20)	4	4	4	4
Sailing (20)	20	20	20	20
Settlers (20)	4	4	4	4
TPP (20)	20	20	20	20
Twin Prime (20)	20	20	20	20
Zenotravel (20)	20	20	20	20
$\sum$ (400)	302	302	307	**308**

Table 1: Max coverage per-domain of each search algorithm on the standard (8 GB RAM) setting

Domain	GBFS		GONDOR	
	no-BF	BF	no-BF	BF
Block Grouping (200)	163	158	161	155
Counters (200)	78	78	97	98
Delivery (200)	143	143	144	144
Drone (200)	149	154	149	157
Expedition (200)	25	28	22	25
Farming (200)	178	178	178	178
FO-Counters (200)	53	53	54	55
FO-Farming (200)	180	180	180	180
FO-Sailing (200)	145	149	157	155
Hydropower (200)	123	140	124	128
Market Trader (200)	127	127	133	133
Pacman (200)	137	140	143	145
Pathways (200)	8	8	8	9
Plant Watering (200)	177	187	184	191
Rover (200)	20	20	20	20
Sailing (200)	147	149	151	152
Settlers (200)	23	23	24	23
TPP (200)	133	132	134	133
Twin Prime (200)	172	172	170	170
Zenotravel (200)	158	158	152	155
$\sum$ (4000)	2339	2377	2385	2406

Table 2: Sum coverage per-domain of each search algorithm on the standard (8 GB RAM) setting

Domain	GBFS		GONDOR	
	no-BF	BF	no-BF	BF
Block Grouping (20)	19	19	19	19
Counters (20)	8	9	10	10
Delivery (20)	18	18	18	18
Drone (20)	16	16	17	17
Expedition (20)	3	3	3	3
Farming (20)	20	20	20	20
FO-Counters (20)	6	6	6	6
FO-Farming (20)	20	20	20	20
FO-Sailing (20)	20	20	20	20
Hydropower (20)	20	20	20	20
Market Trader (20)	20	20	20	20
Pacman (20)	16	16	16	16
Pathways (20)	1	1	1	1
Plant Watering (20)	17	20	20	20
Rover (20)	4	4	4	4
Sailing (20)	20	20	20	20
Settlers (20)	4	4	4	4
TPP (20)	13	14	18	20
Twin Prime (20)	20	20	20	20
Zenotravel (20)	20	20	20	20
$\sum$ (400)	285	290	296	**298**

Table 3: Max coverage per-domain of each search algorithm on the low memory (512 MB) setting.

Domain	GBFS		GONDOR	
	no-BF	BF	no-BF	BF
Block Grouping (200)	148	150	140	156
Counters (200)	68	73	73	78
Delivery (200)	143	143	146	143
Drone (200)	139	139	145	145
Expedition (200)	17	20	18	19
Farming (200)	169	169	170	171
FO-Counters (200)	45	45	51	51
FO-Farming (200)	171	171	176	180
FO-Sailing (200)	127	128	133	136
Hydropower (200)	108	112	108	112
Market Trader (200)	104	104	117	122
Pacman (200)	116	122	125	129
Pathways (200)	7	8	8	8
Plant Watering (200)	89	118	106	154
Rover (200)	20	20	20	20
Sailing (200)	115	121	125	127
Settlers (200)	14	13	21	23
TPP (200)	79	81	91	103
Twin Prime (200)	172	172	171	170
Zenotravel (200)	149	151	155	149
$\sum$ (4000)	2000	2060	2099	2196

Table 4: Sum coverage per-domain of each search algorithm on the low memory (512 MB) setting.

scatterplots

Below are the scatterplots of run time, expanded nodes, generated nodes, and solution length of $GONDOR_{BF}$ as well as $GONDOR$ and $GBFS_{BF}$ against the baseline. We note that expanded and generated notes are not directly comparable between GONDOR and non-GONDOR methods since GONDOR may expand or generate the same node multiple times. We nevertheless present them here for completion's sake.

For clarity, all runtime plots are scaled to ignore runs shorter than 1s, and all expanded and generated nodes plots are scaled to 1000+.

Some artifacts appear in expanded nodes = 1000, 2000, 3000. These are logging artifacts, as for failures we only log every 1000 expansions.

run times

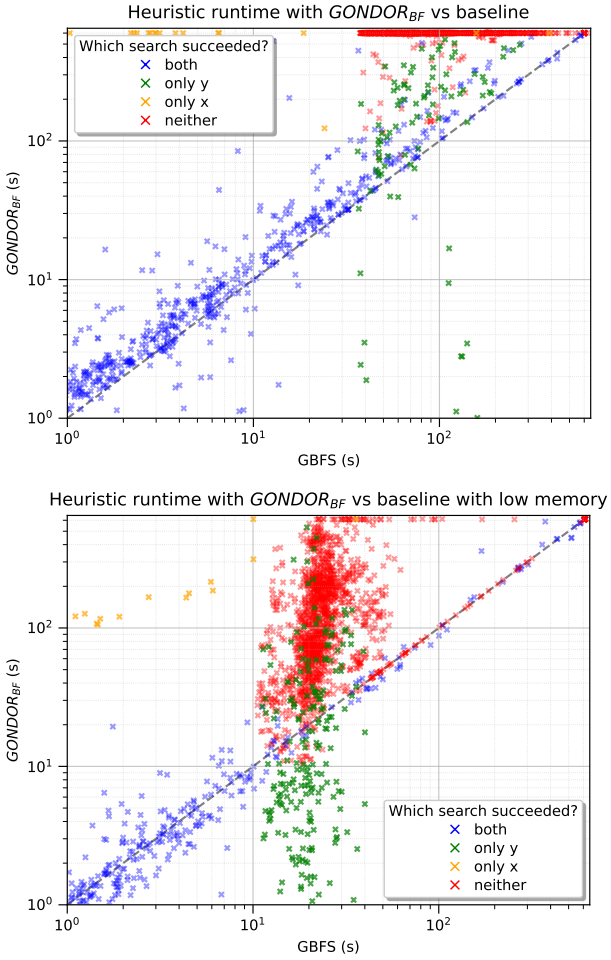

Figure 1: Runtime comparison of each heuristic on each instance of $GONDOR_{BF}$ vs the $GBFS$ baseline. Each X represents one search with each method, and is colored by which search algorithm found a solution successfully. $GONDOR_{BF}$ wins for points below the diagonal Points below the diagonal and points in green, and loses above the diagonal and for points in orange. The same experiment is repeated with 8GB RAM (top) and 512 MB (bottom)

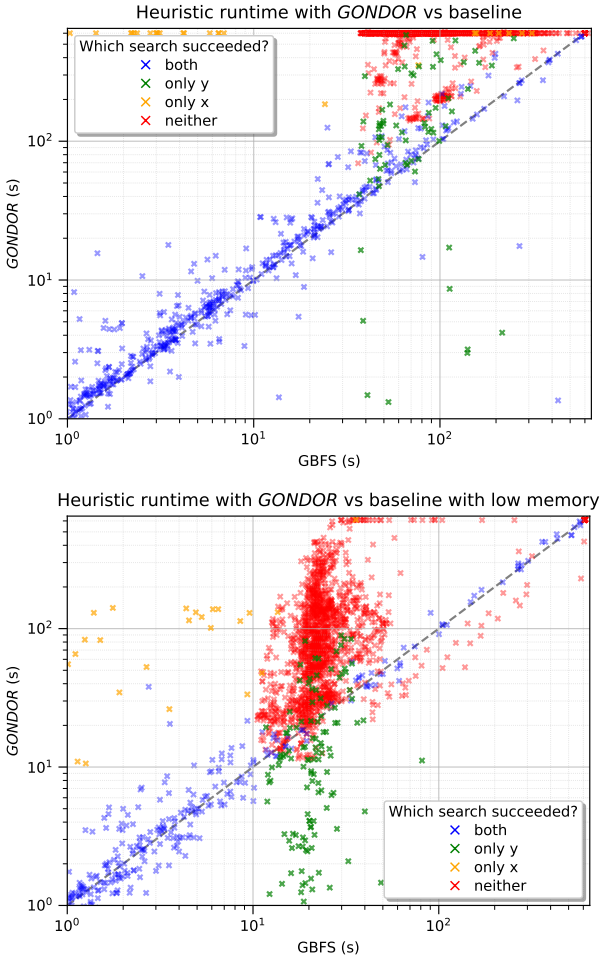

Figure 2: Runtime comparison of each heuristic on each instance of $GONDOR$ vs the $GBFS$ baseline. Each X represents one search with each method, and is colored by which search algorithm found a solution successfully. $GONDOR$ wins for points below the diagonal Points below the diagonal and points in green, and loses above the diagonal and for points in orange. The same experiment is repeated with 8GB RAM (top) and 512 MB (bottom)

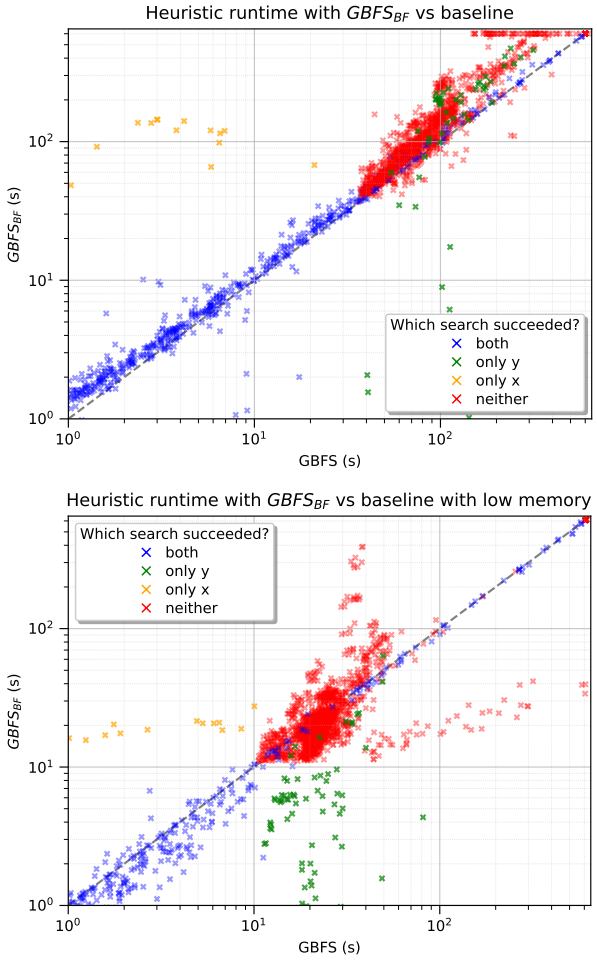

Figure 3: Runtime comparison of each heuristic on each instance of $GBFS_{BF}$ vs the $GBFS$ baseline. Each X represents one search with each method, and is colored by which search algorithm found a solution successfully. $GBFS_{BF}$ wins for points below the diagonal Points below the diagonal and points in green, and loses above the diagonal and for points in orange. The same experiment is repeated with 8GB RAM (top) and 512 MB (bottom)

expanded nodes

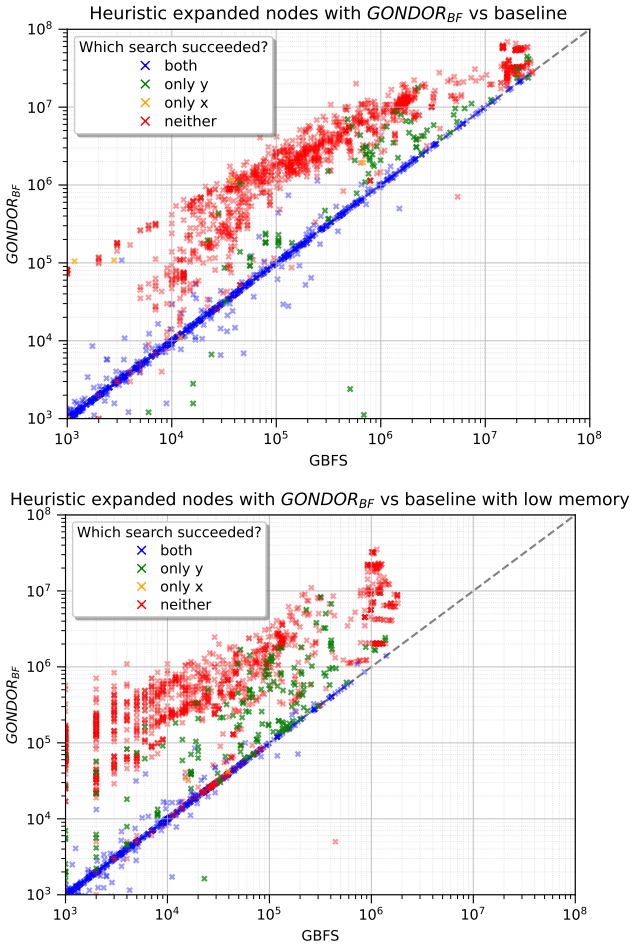

Figure 4: Expanded nodes comparison of each heuristic on each instance of $GONDOR_{BF}$ vs the $GBFS$ baseline. Each X represents one search with each method, and is colored by which search algorithm found a solution successfully. $GONDOR_{BF}$ wins for points below the diagonal Points below the diagonal and points in green, and loses above the diagonal and for points in orange. The same experiment is repeated with 8GB RAM (top) and 512 MB (bottom)

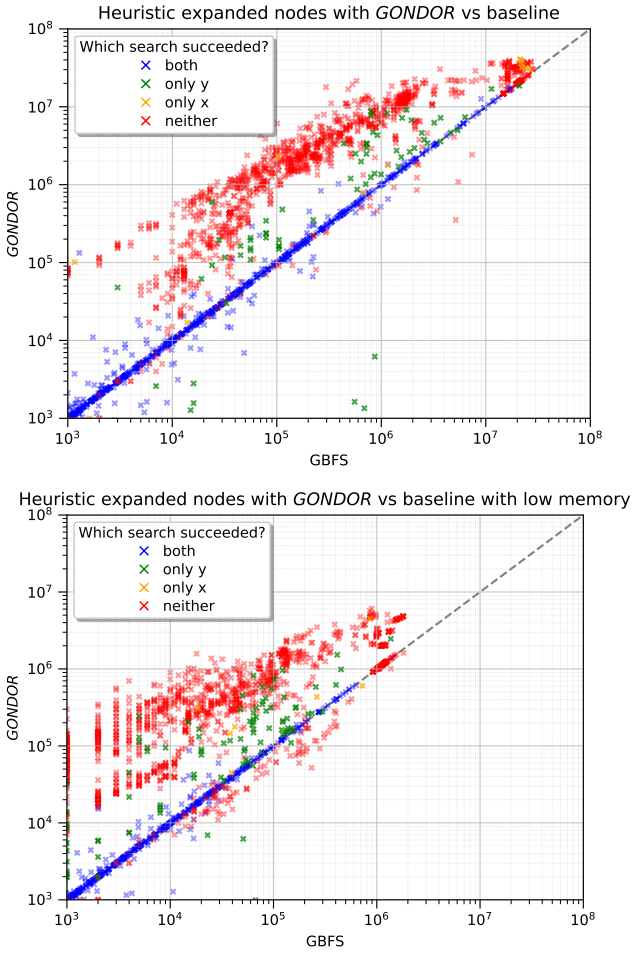

Figure 5: Expanded nodes comparison of each heuristic on each instance of $GONDOR$ vs the $GBFS$ baseline. Each X represents one search with each method, and is colored by which search algorithm found a solution successfully. $GONDOR$ wins for points below the diagonal Points below the diagonal and points in green, and loses above the diagonal and for points in orange. The same experiment is repeated with 8GB RAM (top) and 512 MB (bottom)

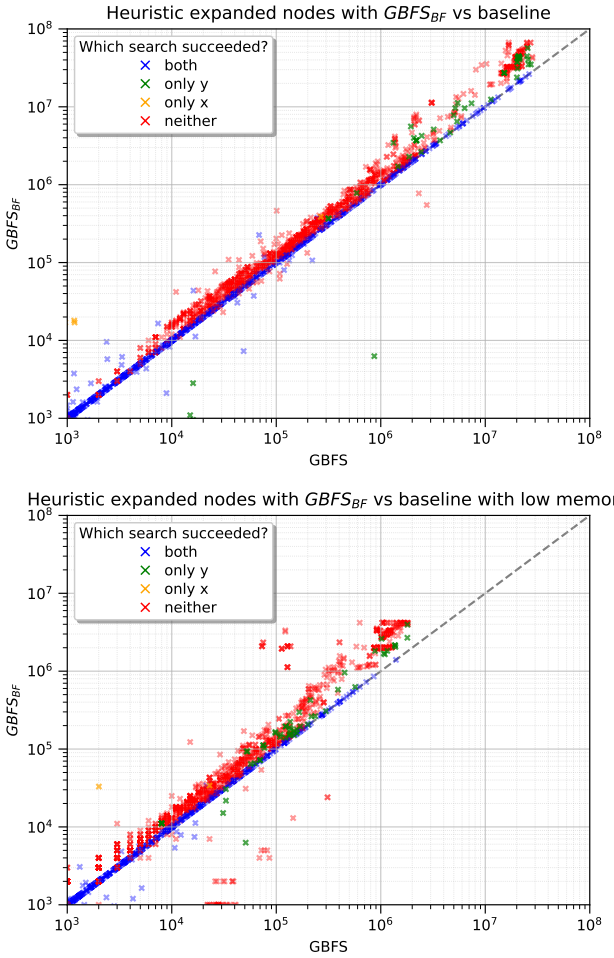

Figure 6: Expanded nodes comparison of each heuristic on each instance of $GBFS_{BF}$ vs the $GBFS$ baseline. Each X represents one search with each method, and is colored by which search algorithm found a solution successfully. $GBFS_{BF}$ wins for points below the diagonal Points below the diagonal and points in green, and loses above the diagonal and for points in orange. The same experiment is repeated with 8GB RAM (top) and 512 MB (bottom)

Figure 7: Generated nodes comparison of each heuristic on each instance of $GONDOR_{BF}$ vs the $GBFS$ baseline. Each X represents one search with each method, and is colored by which search algorithm found a solution successfully. $GONDOR_{BF}$ wins for points below the diagonal Points below the diagonal and points in green, and loses above the diagonal and for points in orange. The same experiment is repeated with 8GB RAM (top) and 512 MB (bottom)

generated nodes

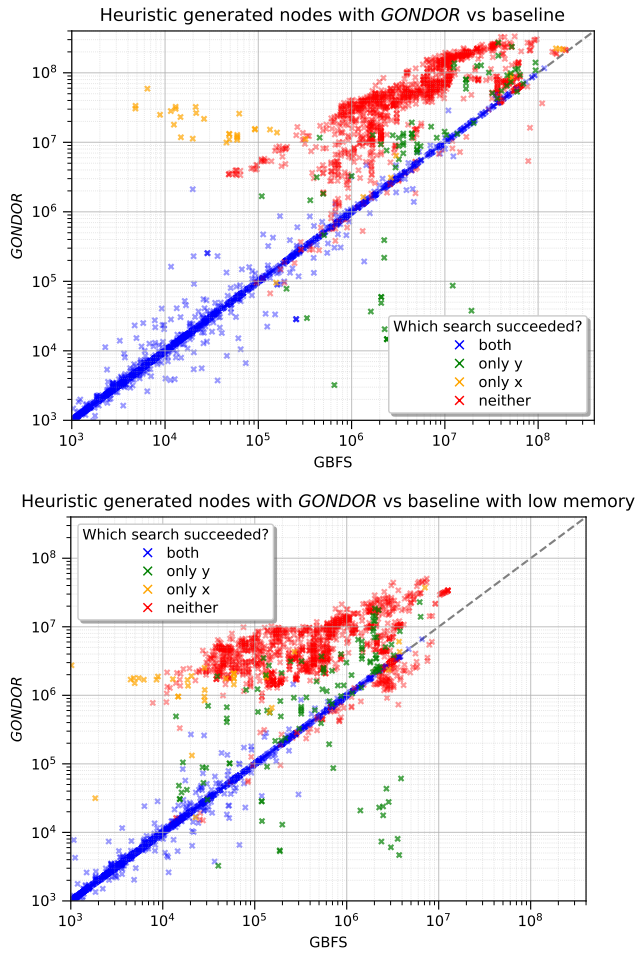

Figure 8: Generated nodes comparison of each heuristic on each instance of $GONDOR$ vs the $GBFS$ baseline. Each X represents one search with each method, and is colored by which search algorithm found a solution successfully. $GONDOR$ wins for points below the diagonal Points below the diagonal and points in green, and loses above the diagonal and for points in orange. The same experiment is repeated with 8GB RAM (top) and 512 MB (bottom)

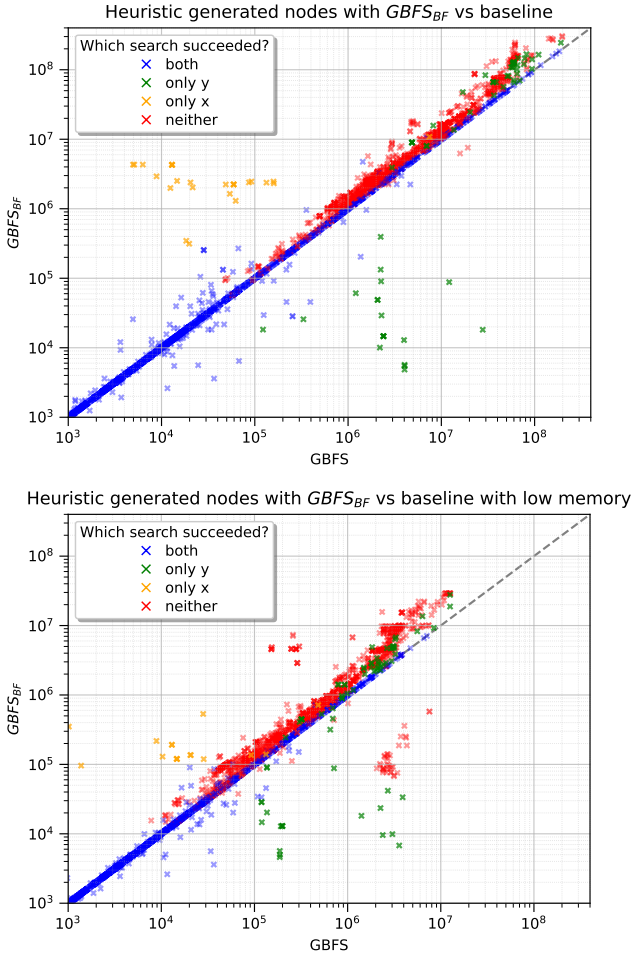

Figure 9: Generated nodes comparison of each heuristic on each instance of $GBFS_{BF}$ vs the $GBFS$ baseline. Each X represents one search with each method, and is colored by which search algorithm found a solution successfully. $GBFS_{BF}$ wins for points below the diagonal Points below the diagonal and points in green, and loses above the diagonal and for points in orange. The same experiment is repeated with 8GB RAM (top) and 512 MB (bottom)

solution lengths

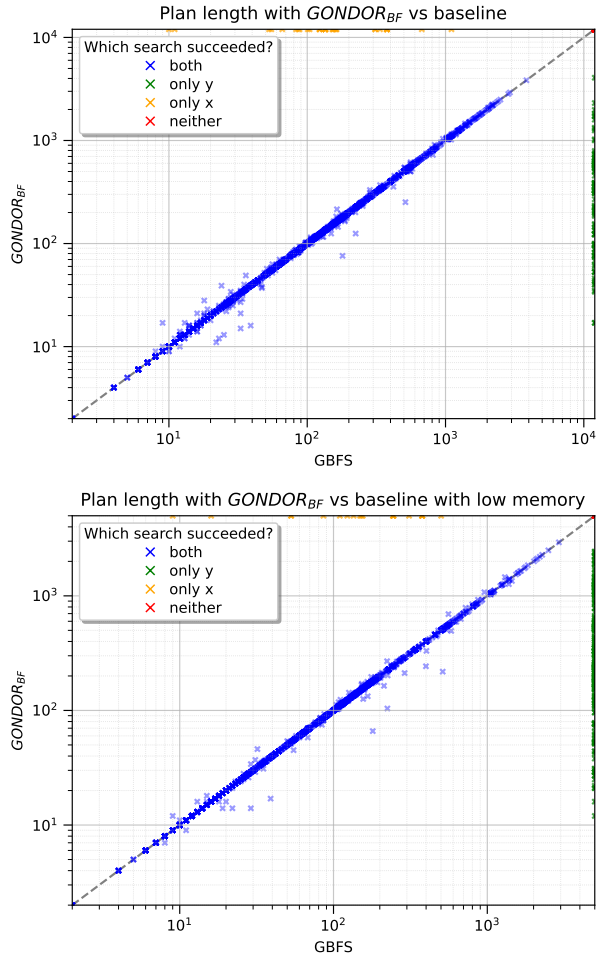

Figure 10: Solution length comparison of each heuristic on each instance of $GONDOR_{BF}$ vs the $GBFS$ baseline. Each X represents one search with each method, and is colored by which search algorithm found a solution successfully, while failed searches are clipped to max. $GONDOR_{BF}$ wins for points below the diagonal Points below the diagonal and points in green, and loses above the diagonal and for points in orange. The same experiment is repeated with 8GB RAM (top) and 512 MB (bottom)

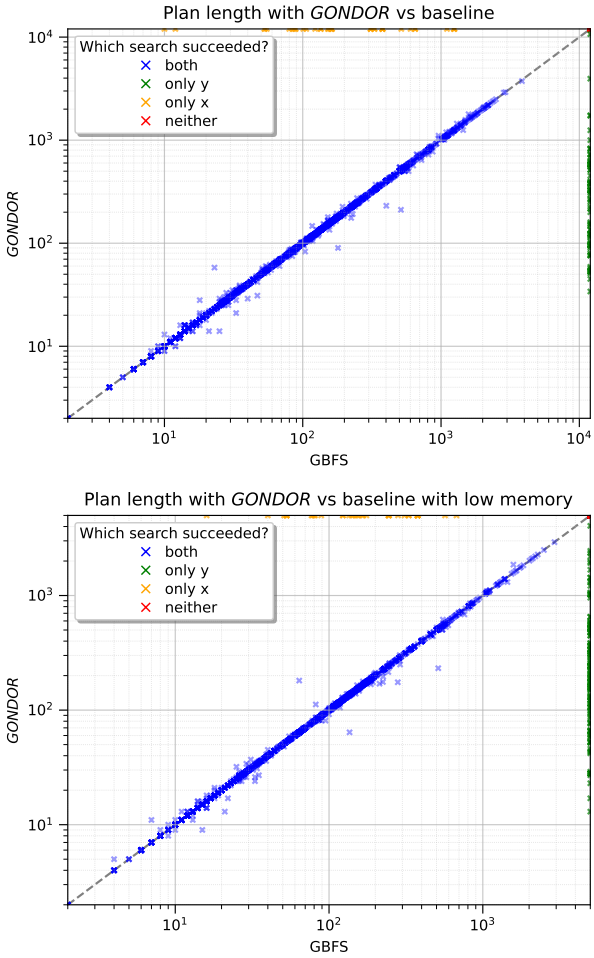

Figure 11: Solution length comparison of each heuristic on each instance of $GONDOR$ vs the $GBFS$ baseline. Each X represents one search with each method, and is colored by which search algorithm found a solution successfully, while failed searches are clipped to max. $GONDOR$ wins for points below the diagonal Points below the diagonal and points in green, and loses above the diagonal and for points in orange. The same experiment is repeated with 8GB RAM (top) and 512 MB (bottom)

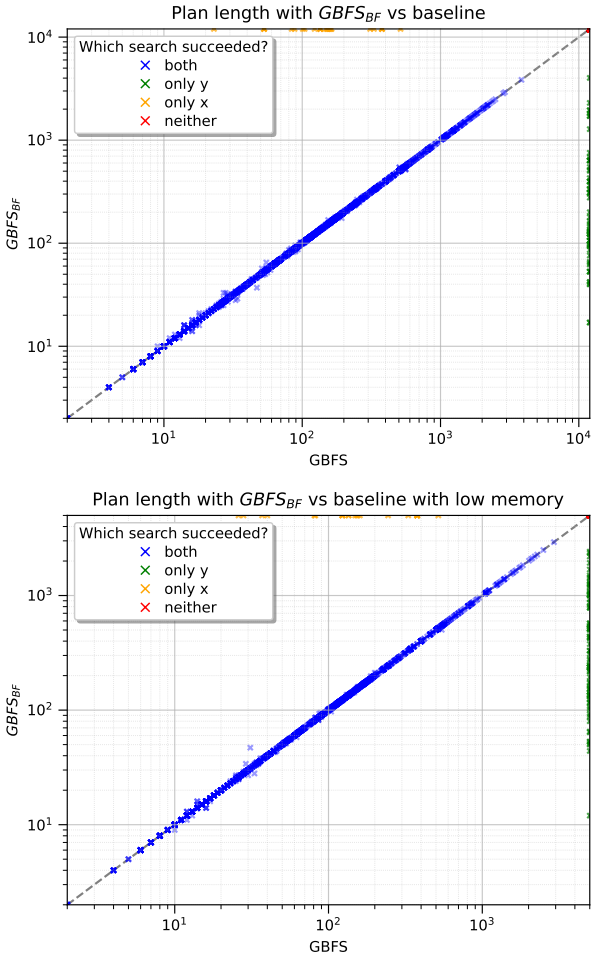

Figure 12: Solution length comparison of each heuristic on each instance of $GBFS_{BF}$ vs the $GBFS$ baseline. Each X represents one search with each method, and is colored by which search algorithm found a solution successfully, while failed searches are clipped to max. $GBFS_{BF}$ wins for points below the diagonal Points below the diagonal and points in green, and loses above the diagonal and for points in orange. The same experiment is repeated with 8GB RAM (top) and 512 MB (bottom)

References

Sievers, S.; and Wehrle, M. 2021. On Weak Stubborn Sets in Classical Planning. In Zhou, Z.-H., ed., *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, 4167–4174. International Joint Conferences on Artificial Intelligence Organization. Main Track.

Valmari, A. 1991. Stubborn sets for reduced state space generation. In Rozenberg, G., ed., *Advances in Petri Nets 1990*, 491–515. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-46369-6.