Adding Exploration to Greedy Best-First Search

Richard Valenzano, Nathan Sturtevant, and Jonathan Schaeffer

Abstract

While greedy best-first search (GBFS) is a popular algorithm for solving automated planning tasks, it can exhibit poor performance if the heuristic in use mistakenly identifies a region of the search space as promising. In such cases, the way the algorithm greedily trusts the heuristic can cause the algorithm to get stuck performing a lot of unnecessary work. In this report, we consider simple techniques that help GBFS to avoid overly trusting the heuristic. The first technique, *heuristic perturbation*, will be shown to lead to large increases in coverage in some domains, and large decreases in others. Over all problems tested, this technique does increase the average coverage by up to 9.5% over standard GBFS when it is parameterized effectively. The second technique, *ϵ-greedy node selection*, will be shown to lead to smaller improvements than heuristic perturbation in many domains, though it does so without hurting the algorithm's performance in any other domains. Over all tested problems, this technique will be shown to increase coverage even when used with a wide range of parameter values, with the best setting leading to a 11.0% increase in coverage when compared to standard GBFS. We will also show that these techniques can be paired together effectively in an algorithm portfolio due to the complementary way they introduce exploration into the search, with our best portfolio having an expected coverage that is 22.5% higher than standard GBFS.

1 Introduction

Greedy Best-First Search (GBFS) is of the most popular heuristic search algorithms used for solving planning tasks which cannot be feasibly solved optimally without a too resource-intensive search. While it has no provable bounds on solution quality, it is typically much faster than optimal algorithms such as A* [4] and algorithms with suboptimality guarantees such as WA* [12]. This advantage can be attributed to the greedy fashion in which GBFS employs the heuristic, unlike A* and WA* which do not use the heuristic information as greedily.

Unfortunately, the greediness of GBFS can also lead to poor behaviour due to error in the heuristic function. This is because GBFS will *trust* or *exploit* the heuristic information completely, and so if the heuristic function mistakenly identifies a region of the domain as promising, the greediness of GBFS can often cause the algorithm to exhaustively search such regions. This suggests the need for techniques that encourage *exploration* in the search performed by GBFS.

In this document, we look at simple ways to do just that. Specifically, we will consider stochastic methods which will cause the algorithm to periodically consider nodes that the heuristic does not identify as being the most promising of those under consideration. We then show empirically that when used in an

```
GBFS(Initial Node n_start):
 1: CLOSED ← {}
 2: OPEN ← {n_start}
 3: loop
 4:   if OPEN is empty then
 5:     return no solution exists
 6:   n ← arg min_{n∈OPEN} h(n)
 7:   if n is a goal node then
 8:     return path from n_start to n
 9:   generate children nodes L_n = {c_1, ..., c_k} of n
10:   for all c ∈ L_n do
11:     if c ∉ OPEN and c ∉ CLOSED then
12:       Add c to OPEN
13:   Remove n from OPEN and add it to CLOSED
```

Algorithm 1: The Best-First Search Algorithm GBFS.

automated planner, the new techniques can improve performance. The first of these techniques, *heuristic perturbation*, will be shown to have different strengths and weaknesses on a domain-by-domain basis, though the overall trend is that more problems are solved when this technique is used. The second of these techniques, *ϵ-greedy node selection*, will also be shown to lead to performance improvements in many domains, though without the loss in performance seen in other domains when using heuristic perturbation. We will also show that by combining these techniques in an *algorithm portfolio*, we can benefit from the strong positive gains made when using heuristic perturbation in some domains with the robust performance seen when using ϵ-greedy node selection in the others.

This report is organized as follows: we begin by first describing the standard GBFS algorithm and the new techniques for encouraging exploration. This is followed by an experimental study into how these techniques impact the performance of GBFS in planning domains, and how they perform when used in a portfolio. This is followed by a discussion of future work and an appendix which provides further detail regarding why heuristic perturbation fails in one particular domain.

2 Greedy Best-First Search

GBFS is an instance of the well-known iterative framework *best-first search*, which uses a *heuristic function*, h, to guide a search for a solution path. This function provides an estimate of the length of the path from any given state to the nearest goal state and it is used to order nodes in terms of how promising they are (*ie.* how quickly they are expected to lead to a goal).

Pseudocode for GBFS is shown in Algorithm 1. The algorithm uses two lists: OPEN and CLOSED. OPEN contains the last node on each of the candidate paths currently being considered, while CLOSED contains all previously examined nodes. The algorithm iteratively selects the node on OPEN with the lowest h-cost for *expansion* (line 6), with ties being broken arbitrarily. This process involves the generation of the *successors* or *children* of that node (line 9). Any children whose corresponding state has been generated for the first time is then added to OPEN

(line 12). Once this is completed, the expanded node is moved to CLOSED (line 13). This process is then continued until either a goal node is expanded (line 7), or the OPEN list is emptied (line 4), in which case there is provably no solution. When a goal node is expanded, the solution path can be extracted from the CLOSED list, which maintains structures for doing just that (see [2] for more detail).

Unlike most best-first search definitions (such as in [2] and [17]), we do not move nodes from CLOSED back to OPEN if a better path is found to them. This *re-opening* step is typically done for the sake of solution quality. As our focus is on planner *coverage* — which is defined as the number of problems solved by an algorithm — we will not perform such updates. This approach is also taken by such planners as LAMA [14] and Fast Downward [5].

In the next section we will consider how this basic version of GBFS can be supplemented so that exploration is added to the search.

3 Adding Exploration to GBFS

In this section, we will consider two techniques for encouraging exploration into GBFS: *heuristic perturbation* and *ϵ-greedy node selection*.

3.1 Heuristic Perturbation

Heuristic perturbation helps GBFS to avoid overly trusting the heuristic by running the search on a noisy version of the given heuristic function. This means that instead of using the heuristic h to guide the search, a GBFS instance using heuristic perturbation will use the heuristic function h_{HP}, defined as $h_{HP}(n) = h(n) + r(n)$ where $r(n)$ is a random integer from the range $[0, a]$ and a is a user set algorithm parameter. We will refer to $h(n)$ as the *non-noisy* heuristic value of n, $r(n)$ as the *noise value of* n, and the parameter a as the *noise level* of the algorithm.

Heuristic perturbation changes the order in which nodes are considered for expansion by changing how promising nodes appear to be. For example, suppose that a node n has a low non-noisy heuristic value. If n is assigned a high noise value, it will appear to be much less promising than it would have otherwise. In such cases, a node with a higher heuristic value which was assigned a low noise value may be picked first.

However, heuristic perturbation is still biased towards nodes with low heuristic values. For example, suppose that the noise level is 4, and that at some time during the search, the lowest non-noisy heuristic value of any of the nodes on the OPEN list is 10. This means that the next node selected must have a non-noisy heuristic value of 14 or lower. Since the expected noise value for any node is 2, and the variance is the same for all nodes, the nodes with the highest probability of being selected are also still those with a non-noisy value of 10.

In the implementation of heuristic perturbation used below, the noise value of a node n will be determined when a node n is first generated. The noise value then remains static for the remainder of the time that n is on OPEN. This decision was made in the interest of algorithm simplicity.

3.2 ϵ-greedy Node Selection

While heuristic perturbation can change how promising nodes appear, *ϵ-greedy node selection* uses an even more explicit way to introduce exploration. This is accomplished by modifying the way nodes are selected from OPEN in line 6 of Algorithm 1. This modification requires the user to set a parameter, denoted ϵ, as some value in the range $[0, 1]$. With probability $(1 - \epsilon)$, ϵ-greedy node selection uses the same rule as standard GBFS: it selects the node on OPEN with the lowest heuristic value, with ties being broken arbitrarily. However, with probability ϵ, this new technique selects a node uniformly at random from amongst all of the nodes in OPEN. This means that with probability $(1-\epsilon)$ this node selection policy chooses a node greedily according to the heuristic, while with probability ϵ the policy selects a node with the aim of exploring.

4 Experiments

In this section, we will experiment with the techniques described above and show that they often lead to improved coverage in planning domains. To do so, we implemented these techniques in the Fast Downward planning system [5] and then ran experiments on a cluster of machines, each with two 4-core 2.8 GHz Intel Xeon E546s processors with 6 MB of L2 cache. The problem set used is given by all 790 tasks from the 2006, 2008, and 2011 International Planning Competitions. On each problem, each planner is allowed a maximum of 30 minutes and 4 GB. As the techniques we are considering are stochastic, we run each tested planner configuration for 10 times per problem and consider the frequency with which a problem is solved as the configuration's coverage on that problem.

As our focus is on coverage, we also treat all tasks as unit-cost tasks. This is consistent with several existing state-of-the-art planners such as LAMA [14] and ArvandHerd [18] which do the same with the goal of maximizing coverage.

4.1 Heuristic Perturbation in a Simple Planner

So as to isolate the effects of the new exploration encouraging techniques, we test them in a simple planner which only uses a single heuristic — the inadmissible FF heuristic [7] — and a common planning enhancement called *deferred heuristic evaluation* [13]. When using this enhancement, the heuristic of a node is only calculated when it is expanded, not when it is generated as is typically done in standard GBFS. This means that when a node n is generated and added to OPEN, the heuristic value used to establish the priority that n has in OPEN is not $h(n)$ — since $h(n)$ will not be calculated until n has been expanded — but the heuristic value of the parent of n (*ie.* the node p whose expansion generated n). As an example of how this works in practice, consider heuristic perturbation. When using heuristic perturbation without deferred heuristic evaluation, the heuristic used to order OPEN is

$$h_{HP}(n) = h(n) + r(n)$$

However, when deferred heuristic evaluation is used, the heuristic looks as follows:

$$h_{HP}(n) = h(p) + r(n)$$

where p is the parent of n.

The average coverage on a per domain basis for different noise levels is shown in Table 1. The table has been portioned off into 3 sections. The first shows the 2006 domains, the second shows the 2008 domains, and the third shows the 2011 domains. The column labelled as using a noise level of 0 is equivalent to using GBFS without heuristic perturbation. The variance in coverage was induced through the use of *random operator ordering* which changes how ties are broken among nodes with equal heuristic values [19].

The last row of the table shows the totals for each configuration tested over all problems. It shows that for noise levels of 16, 64, and 256, heuristic perturbation is able to solve more problems than the number solved without heuristic perturbation. GBFS with heuristic perturbation at a noise level of 1 has worse coverage overall when compared to not using any noise, while noise levels of 2 and 4 have similar coverage to that seen without any noise.

The table also shows that heuristic perturbation often leads to substantial changes in coverage — either increases or losses — in several domains. For example, this technique is quite helpful in 2011 barman, 2011 visitall, and 2011 woodworking, in which standard GBFS solves a total of 17.8 problems on average, GBFS at a noise level of 1 solves a total of 22.8 problems on average, and GBFS at a noise level of 256 solves a total of 56.2 problems on average. In other domains, such as 2011 parcprinter, 2011 parking, and 2006 openstacks, the opposite behaviour emerges. In these domains, standard GBFS solves a total of 56 problems on average, GBFS at a noise level of 1 solves a total of 35.2 problems on average, and GBFS at a noise level of 256 solves a total of 11.3 problems on average. In Section 4.3, we will show that this variance across domains can be leveraged effectively in the construction of portfolio-based systems.

The 2006 openstacks domain is notable since standard GBFS solves all 30 problems in all runs, while the use of heuristic perturbation causes a drastic drop in coverage even when the noise level is 1. GBFS using heuristic perturbation at a the noise level of 1 would actually outperform standard GBFS by an average of 10.3 problems if the 2006 openstacks domain is omitted from the results. We consider this domain in more detail in the appendix.

4.2 ϵ-Greedy Node Selection in a Simple Planner

In this section, we consider the performance of ϵ-greedy node selection. As with heuristic perturbation, the FF heuristic and deferred heuristic evaluation were used. The average coverage on a per domain basis is shown for different noise levels in Table 2. The table shows that with the exception of very high values for ϵ, ϵ-greedy node selection improves the overall coverage of GBFS even more so than heuristic perturbation.

In contrast to heuristic perturbation, ϵ-greedy node selection typically results in much smaller performance changes on a domain-by-domain basis. There are no domains in which ϵ-greedy node selection leads to any more than a minor decrease in coverage except when ϵ is almost 1. In domains in which this technique does lead to coverage gains, it is usually unable to increase it by as much as heuristic perturbation. The 2011 woodworking domain is a clear example of this, as the coverage does increase from the 2.6 problems solved on average when using standard GBFS to 13.5 when $\epsilon = 0.5$, but it never gets as high as the 20 seen when using

Domain Name	# of Probs	Noise Level 0	1	2	4	16	64	256
2006 openstacks	30	30	7.4	7.6	7.3	7.9	9	10.5
2006 pathways	30	9.2	9.1	9.7	7.4	4.7	4.8	5.1
2006 rovers	40	22.3	22.9	23.2	23.8	24.3	29	32.2
2006 storage	30	20.1	19.3	20.4	21.8	22.2	23.8	21.6
2006 tankage	50	21.4	22.3	22.4	22.4	26	33.9	38.1
2006 tpp	30	21.4	22	23.3	24.6	30	30	30
2006 trucks	30	16.1	15.1	15.4	15.5	16.5	19	18.8
2006 Totals	240	140.5	118.1	122	122.8	131.6	149.5	156.3
2008 cybersec	30	25.3	26.8	26.8	28.7	29.1	29.4	17.9
2008 elevators	30	30	30	30	30	30	30	30
2008 openstacks	30	30	30	30	30	30	30	30
2008 parcprinter	30	25.7	24.9	24.4	21.8	20.9	18.4	14.5
2008 pegsol	30	30	30	30	30	30	29.7	29.1
2008 scanalyzer	30	27.6	27.7	27.4	28.1	29.6	27.3	24.1
2008 sokoban	30	29	29	29	29	28.4	28.4	27.1
2008 transport	30	17.2	17.6	17.9	16.9	19.4	21.1	24.3
2008 woodworking	30	15.4	15.7	16	16.5	21.9	30	30
2008 Totals	270	230.2	231.7	231.5	231	239.3	244.3	227
2011 barman	20	11.4	15.7	18.9	19.7	19.9	20	18
2011 elevators	20	13.3	13.9	15.2	15.1	17.2	20	20
2011 floortile	20	4.2	4.4	4.4	4.7	4.6	5	6.6
2011 nomystery	20	8.4	6.2	5.8	5.7	5.5	5.6	4.9
2011 openstacks	20	18.5	20	20	20	20	19.9	19.7
2011 parcprinter	20	11.6	10.9	11.1	9.7	7.4	4.1	0.6
2011 parking	20	14.4	16.9	16.9	16.5	7.2	2.4	0.2
2011 pegsol	20	20	20	20	20	20	19.9	19.4
2011 scanalyzer	20	17.8	17.9	17.7	17.8	19.4	17.6	14.4
2011 sokoban	20	19	19	19	19	18.7	18.4	16.9
2011 tidybot	20	11.2	13.1	14.2	13.8	16.6	16.4	11.8
2011 transport	20	0	0	0	0.1	0.6	2.6	4.7
2011 visitall	20	3.8	4.6	5.4	6.3	8.7	12.5	18.2
2011 woodworking	20	2.6	2.2	2.5	2.9	6.8	19	20
2011 Totals	280	156.2	164.8	171.1	171.3	172.6	183.4	175.4
All Domains Totals	790	526.9	514.6	524.6	525.1	543.5	577.2	558.7

Table 1: The coverage of heuristic perturbation with lazy heuristic evaluation.

heuristic perturbation at a noise level of 256. However, as these more modest coverage improvements do not come with coverage losses in other domains, ϵ-greedy node selection did lead to a higher coverage than heuristic perturbation.

There is some correlation between the domains in which heuristic perturbation and ϵ-greedy node selection node selection result in coverage improvements. For example, both lead to improvements 2006 tankage, 2011 barman, 2008 woodworking, and 2011 woodworking, though heuristic perturbation typically yields larger coverage gains in such domains. However, there are domains in which ϵ-greedy node selection leads to modest improvements while heuristic perturbation leads to no change in coverage or a decrease in coverage. For example, in 2011 floortile, 2011 nomystery, and 2011 parcprinter, standard GBFS solves a total of 24.2 on average, while $\epsilon = 0.3$ solves a total of 29 on average, and GBFS at a noise level of 64 (the noise level with the highest overall coverage) solved a total of 14.7 on average. There are also domains in which heuristic perturbation improves coverage and ϵ-greedy node selection does not. 2006 tpp and 2008 transport are two such examples. In these domains, both standard GBFS and $\epsilon = 0.2$ solve an average of 38.6 problems, while heuristic perturbation at a noise level of 256 solves 51.1.

Since ϵ-greedy node selection often leads to modest coverage gains on a domain-by-domain basis without losing much coverage in other domains, it can be viewed as a *low-risk* technique. This is because there is little risk in supplementing GBFS with ϵ-greedy node selection and badly hurting performance. In contrast, heuristic perturbation when using a high noise level can be seen as *high-risk* since the chance of it significantly improving GBFS also comes with a chance of it causing a substantial decrease in performance. In the next section, we will show that by combining these low and high-risk techniques in an algorithm portfolio, we can benefit from the strengths of both.

4.3 Using GBFS Techniques in a Portfolio

An *algorithm portfolio* is a collection of planning techniques that are available to a single planning system for use on any given problem. Once a portfolio of algorithms has been selected, it can be used in a variety of ways. One popular way is to use a classifier to select a single planner from the portfolio for a given problem, based on features of the problem description [15, 1]. While this can be effective in practice, it does require a substantial training phase. A second technique is to use all planners in the portfolio on the given problem by running each in turn for some portion of the available time [6, 18, 16]. An easy way to do this is to assign each an equal portion of the planning time. For example, if there are k planners in the portfolio and the time limit is 30 minutes, we can run each for 30/k minutes. This approach will be referred to as *uniform time partitioning* [18].

Below we will show that these new exploration encouraging techniques lead to even larger coverage gains when they are used in a portfolio. To do so, we will use the empirical results summarized in tables 1 and 2 to estimate the expected performance of a number of different portfolios which are each deployed using uniform time partitioning. For details on how these calculations are performed, see [18]. The results are shown in Table 3 which considers portfolios, mostly of size 2. The candidates for the portfolios included standard GBFS, and 4 parameter settings of each of the heuristic perturbation and ϵ-greedy approaches.

Domain Name	# of Probs	ϵ 0.0	0.1	0.2	0.3	0.5	0.75	0.99
2006 openstacks	30	30	29.9	29.8	29.9	30	29.4	29.3
2006 pathways	30	9.2	11.2	11.3	11.4	10.1	10	5.7
2006 rovers	40	22.3	25.4	25.9	25.8	26.1	24.2	18.1
2006 storage	30	20.1	20.9	20.8	20.8	21	20.8	18.6
2006 tankage	50	21.4	25.7	26.5	26.6	26.6	26.9	21.6
2006 tpp	30	21.4	21.8	21.1	20.1	17.6	16.1	13.1
2006 trucks	30	16.1	18.2	18.1	17.6	17.8	16.9	15.2
2006 Totals	240	140.5	153.1	153.5	152.2	149.2	144.3	121.6
2008 cybersec	30	25.3	29.9	29.4	30	29.5	30	29.8
2008 elevators	30	30	30	30	30	30	30	29
2008 openstacks	30	30	30	30	30	30	30	30
2008 parcprinter	30	25.7	26.8	27.2	26.8	26.4	26.5	25.7
2008 pegsol	30	30	30	30	30	30	30	29.9
2008 scanalyzer	30	27.6	29.1	28.7	28.8	27.9	27.5	22.1
2008 sokoban	30	29	29	29	29	29	29	28
2008 transport	30	17.2	17.9	17.5	17.8	16.8	15.8	13.7
2008 woodworking	30	15.4	23.9	26.3	27.3	27.7	24.4	16.1
2008 Totals	270	230.2	246.6	248.1	249.7	247.3	243.2	223.3
2011 barman	20	11.4	17.8	18.2	18.3	17.5	14	0
2011 elevators	20	13.3	14.7	14.7	14.9	13.7	11.9	8.6
2011 floortile	20	4.2	6.4	6.3	6.4	6.5	6.2	5.1
2011 nomystery	20	8.4	9.1	8.6	8.5	9.3	9.1	7.7
2011 openstacks	20	18.5	18.5	17.9	17.5	16.5	14.4	10
2011 parcprinter	20	11.6	13.7	14.1	13.6	13.8	13	12.3
2011 parking	20	14.4	12.5	12.6	11.5	10	6.4	0.3
2011 pegsol	20	20	20	20	20	20	20	19.9
2011 scanalyzer	20	17.8	18.7	18.4	18.6	18.1	17.3	12.1
2011 sokoban	20	19	19	19	19	19	19	18.1
2011 tidybot	20	11.2	13	13.8	14.4	14.6	13.7	6.8
2011 transport	20	0	0.2	0.1	0	0	0	0
2011 visitall	20	3.8	7	6.9	6.6	5.8	4.6	1.8
2011 woodworking	20	2.6	11.1	12.9	13.5	13.2	9.2	2
2011 Totals	280	156.2	181.7	183.5	182.8	178	158.8	104.7
All Domains Totals	790	526.9	581.4	585.1	584.7	574.5	546.3	450.6

Table 2: The coverage of ϵ-greedy node selection with lazy heuristic evaluation.

Each entry in the table shows the expected performance of a different portfolio. For example, the entry in the row labelled "GBFS" and the column labelled "Noise Level 16" shows that a portfolio containing one instance of standard GBFS and one instance of GBFS with heuristic perturbation at a noise level of 16 is expected to solve an average of 588.9 of the 790 problems in our test set. The data in each row therefore shows the expected performance of all portfolios tested which contain the technique corresponding to the row label, with the best portfolio containing that technique shown in bold. The rows marked NL=k refer to GBFS using heuristic perturbation at a noise level of k, while $\epsilon = j$ refers to ϵ-greedy node selection with $\epsilon = j$. The "Alone" column shows the performance of the corresponding technique when it is used alone and not in a portfolio. This data is taken directly from the "All Domains Totals" rows in Tables 1 and 2.

When the column and row refer to the same planner, we show the performance of a portfolio which contains several instances of the same technique which only differ in their random seed. For these entries, we considered every portfolio containing anywhere from 2 to 10 instances, with the best coverage seen by any of these portfolio sizes being shown. The number of planner instances used in the portfolio is shown in brackets. The table shows that in almost all such cases (which can be found along the diagonal of the table), it is at least as good to perform uniform time partitioning over multiple instances of the same technique than it is to let a single run of that technique to use the entire 30 minutes. As described in [18], this is because uniform time partitioning can take advantage of the variance caused by the stochastic nature of all techniques considered.

However, none of the portfolios containing only multiple instances of the same planning technique appear in bold, as in all tested cases it is better to mix-and-match techniques. In particular, the best portfolios appear to be constructed by combining a low-risk technique (such as standard GBFS, GBFS at a low noise level, or ϵ-greedy node selection), with a high-risk technique (such as GBFS with a high noise level). For example, the best portfolios containing standard GBFS or GBFS at the low noise level of 4, are achieved when these are each combined with GBFS instances with noise levels of 64 and 256, respectively. Similarly, the high noise level GBFS instances pair best with ϵ-greedy instances, while the ϵ-greedy instances are best paired with GBFS instances at a high noise level. In particular, the best expected coverage is achieved by the portfolio that contains one instance of GBFS at a noise level of 256 and an instance of GBFS using ϵ-greedy node selection with $\epsilon = 0.2$. This portfolio solves 22.5% more problems than standard GBFS.

In contrast, combining only multiple low-risk approaches or only high-risk approaches is ineffective. This can be seen when two different ϵ-greedy approaches are used in a portfolio, or when combining ϵ-greedy with standard GBFS. Similarly, using a portfolio that only contains GBFS instances at high noise levels also leads to only minor coverage improvements.

5 Future Work

In this section, we describe future work along two lines. In the first, we consider better understanding the techniques described above, while in the second we consider an alternative way to introduce exploration into search.

			Heuristic Perturbation				ϵ-Greedy Node Selection			
	Alone	GBFS	NL=4	NL=16	NL=64	NL=256	$\epsilon = 0.1$	$\epsilon = 0.2$	$\epsilon = 0.3$	$\epsilon = 0.5$
GBFS	526.9	523.1(3)	561.2	588.9	**628.2**	625.0	573.7	580.2	581.6	577.8
NL=4	525.1	561.2	543.7(6)	563.1	603.9	**618.0**	593.6	598.4	599.9	598.1
NL=16	543.5	588.9	563.1	566.4(7)	592.4	608.7	607.2	611.1	**611.5**	609.6
NL=64	577.2	628.2	603.9	592.4	589.5(3)	597.7	**636.2**	636.1	635.5	633.3
NL=256	558.7	625.0	618.0	608.7	597.7	562.5(2)	643.6	**645.2**	644.2	641.2
$\epsilon = 0.1$	581.4	573.7	593.6	607.2	636.2	**643.6**	582.4(2)	588.1	589.1	586.7
$\epsilon = 0.2$	585.1	580.2	598.4	611.1	636.1	**645.2**	588.1	587.6(2)	589.5	587.5
$\epsilon = 0.3$	584.7	581.6	599.9	611.5	635.5	**644.2**	589.1	589.5	587.3(2)	587.1
$\epsilon = 0.5$	574.5	577.8	598.1	609.6	633.3	**641.2**	586.7	587.5	587.1	580.4(2)

Table 3: The expected performance of portfolios constructed by pairing the planning technique shown in the row and the column. The column marked "Alone" shows the performance of each technique when used without a portfolio. When the row and column planner is the same, the table shows the best performance of any portfolio of size 2 to 10 (the best shown in brackets) when restarting multiple times with the same planner.

5.1 Understanding Exploratory GBFS

While the above experiments provide evidence that there are benefits to be had by adding exploration to GBFS, more experimentation is needed to determine how this exploration interacts with other popular planning techniques. State-of-the-art planners such as LAMA use several other planning techniques including *multi-queue best-first search* [5] and *preferred operators* [13]. Exploration may have less of an impact when the planner is strengthened by these other means, and determining which of these planning enhancements still work well with heuristic perturbation and ϵ-greedy node selection remains an important topic for further study.

There are also other existing techniques for adding randomness and diversity into search, such as *UCT* [9], *monte-carlo random-walk planning* [11], *diverse best-first search* [8], and *k-best-first search* [3]. Further study is needed so as to determine the relationship between these and the new techniques proposed in this report.

5.2 Alternative Methods for Introducing Exploration

While ϵ-greedy node selection strategy occasionally samples uniformly from the nodes in OPEN, an alternative approach could be to always sample based on some probabilistic distribution defined over all the nodes in OPEN. This is the idea behind Open List Sampling Search. One possible distribution is given by the *softmax function* for which the probability of selecting a node n is given by

$$\frac{e^{-\lambda h(n)}}{\sum_{n' \in OPEN} e^{-\lambda h(n')}}$$

where λ is a user set parameter such that $\lambda \geq 0$. This function causes node selection to be exponentially biased towards lower heuristic values. However, as λ approaches 0, this distribution "flattens" out and approaches the uniform distribution. An analysis of what sort of distributions lead to effective searches, or which distributions mix well to make an effective portfolio, remain as future work.

6 Conclusion

In this report, we have considered simple techniques for introducing exploration into GBFS. The first of these techniques, heuristic perturbation, was shown to be a high-risk approach that leads to substantial coverage improvements in some domains and substantial coverage losses in others. The second of these techniques, ϵ-greedy node selection, is a much lower-risk approach that leads to modest coverage improvements over a variety of domains without substantial coverage losses in any other domains. However, both techniques did lead to coverage improvements when considered over the entire test set used for a variety of parameter settings.

Since the two techniques achieve their coverage improvements in different ways, they were also shown to pair together well in portfolios. In particular, a portfolio containing one instance of GBFS using ϵ-greedy node selection and one instance of GBFS using heuristic perturbation was able to solve a total of 118.3 more problems on average than standard GBFS alone, which represents a 22.5% increase.

Appendix A Heuristic Perturbation in the 2006 Openstacks Domain

As described earlier, heuristic perturbation is particularly harmful in the 2006 openstacks domain. We will consider why in this appendix.

A.1 Openstacks 2006 Domain Description

In this domain, the task is to schedule the manufacturing of k different types of products so as to satisfy m given orders, each consisting of some subset of the k products. This schedule should be selected such that as little of the storage space on the manufacturing floor is used at all times during the manufacturing process. Manufacturing floor space is taken up by orders which have been partially filled, but for which all required products have not been manufactured. Each order which is currently being filled is given its own *stack* on the manufacturing floor. Such orders are said to have been *started* but not *shipped*. The products necessary for a particular order will be placed on the corresponding stack once they are manufactured. Once all of the products for a particular order have been manufactured, that order can be shipped and its corresponding stack will become available for another order. The best plan will then find an ordering for the manufacturing of the products such that the maximum number of stacks needed at any time during production is minimized.

This process is also constrained by the fact that only one kind of product p can be manufactured at a time, and due to the prohibitive cost of changing production from one product to another, the same product cannot be produced twice until the given set of orders are all satisfied. This means that when the production of product p has been initiated, the total amount of p needed to satisfy all given orders is done at the same time, and each of the orders requiring p must each already have a stack designated to them.

While it is difficult to find the best solution, it is easy to find a solution to a given openstacks problem since any ordering for the manufacturing of the products will suffice. Yet, when heuristic perturbation is added to the Fast Downward planner, the achieved coverage in this domain is quite low. So as to see why, we will need to briefly consider some aspects of how this problem is represented as a planning problem. We do so in the next section.

A.2 Openstacks 2006 Domain Representation

In the representation of this domain used in the planning competition and the experiments above, there is an predicate encoding of a variable called `stacks-avail`. This variable is used to represent the number of stacks that can be assigned to orders without increasing the maximum number of stacks used at any time during production. For example, if the maximum number of stacks needed at any time previously during the manufacturing process is 5 and there are currently only 2 orders which have been started but not shipped, then `stacks-avail` will be 3. This is because at most 3 new orders can be started along with the current 2, before the maximum seen at any time must increase. Note that in the initial state, `stacks-avail` is 0 since no stacks have been needed at that point.

The problem representation also requires that `stacks-avail` never be larger than the total number of orders, m, since there will never need to be any more than m stacks assigned to orders at any time. While this constraint seems rather intuitive, it can lead to problems for searches that include exploration. To see this, we need to consider the actions that can change the value of `stacks-avail`. We begin with the two actions that can increase its value by one. The first is `ship-order`, which takes a completed order as a parameter and ships it. In doing so, this action makes a previously used stack available to be designated to some other order. The second action which increases `stacks-avail` is `open-new-stack`. This action corresponds to a request from the planner for a new unused stack. It is intended to allow the planner to explicitly increase the maximum number of stacks used at any one time. There is also one action which can decrease `stacks-avail`: `start-order`. This action takes as a parameter one of the orders which has not been started or shipped and it decreases the value of `stacks-avail` by designating an unused stack to that order. Since an order can only have a stack designated for it if there is an available stack, `start-order` can only be applied if `stacks-avail` is larger than 0.

Now let us reconsider the constraint which requires that `stacks-avail` is never larger than m. One consequence of this constraint is that both `ship-order` and `open-new-stack` are not applicable if the value of `stacks-avail` is m since applying either would increase `stacks-avail` to $m + 1$. Surprisingly, this also means that despite the fact that any ordering for the manufacturing of the products leads to a goal state, there are also regions of the state space from which no goal state can be reached. To see this, consider a problem in which there are 10 orders, each of which needs the same one product. One of the optimal solutions is to make 10 `open-new-stack` actions, designate each of the newly available stacks to an order, make the product, and then ship each of the completed orders.

Consider the state in this optimal solution which is right after the product has been made but prior to the shipping phase. We will call this state S. `stacks-avail` will be 0 in S, since each of the 10 stacks made available by the application of 10 `open-new-stack` actions has been designated to an order. Now consider the child of S, denoted S', which is the result of applying the unnecessary, but legal action of `open-new-stack` to S. The state is not on the optimal path described above. By applying `open-new-stack` to S, `stacks-avail` is increased from 0 in S to 1 in S'. Since there are no more orders to be started, there are no actions available that will decrease `stacks-avail`. The result is that there are 10 orders to be shipped, but only 9 `ship-order` actions which can be made, since each `ship-order` will increment `stacks-avail` and `stacks-avail` can never be larger than 10. As such, it is not possible to reach a goal state from S' or any of its descendents. We will refer to such areas of the state space *dead-end regions*.

A.3 The FF Heuristic on Openstacks 2006 Problems

In the example given above, the dead-end region is quite large. One reason for this is that there are 10! different ways to ship 9 of the 10 orders, However, if the heuristic identifies that there is no solution possible from S', then this state can be safely pruned and the region will be avoided. Unfortunately, this is not recognized by the FF heuristic which instead indicates that progress is made whenever a `ship-order` action is applied in any descendent of S'. Instead, the FF heuristic will only prune

those descendents of S' in which `stacks-avail` is m.

The FF heuristic is unable to recognize that no goal states can be reached from within these dead-end regions because of the way it evaluates states. This heuristic uses *delete relaxation* to define a simpler planning task for which a solution can be found easily. The length of this solution is then used to estimate the distance from the goal in the non-relaxed problem. In the relaxed problem, a fact which is true in the current state cannot become false. For example, recall that `stacks-avail` is 1 in S'. If a `ship-order` action is made to S' in the relaxed version of this problem, `stacks-avail` is both 1 and 2 in the successor state. This means that regardless of what sequence of actions is applied to the relaxed version of S', the fact that `stacks-avail` is some value other than 10 will always remain true. This allows for all 10 `ship-order` actions to be applied in the relaxation of the problem even though only 9 can be applied in the original problem. The FF heuristic thus incorrectly suggests that goals are reachable from states in the dead-end region.

This behaviour is similar to that seen when using the FF heuristic in *resource-constrained domains*, as shown by Nakhost *et al.* [10]. In the case of openstacks 2006, the resource is the difference between `stacks-avail` and the number of orders m, and both `ship-order` and `open-new-stack` are actions which consume it.

A.4 Avoiding Dead-End Regions Using Standard GBFS

The FF heuristic's inability to detect dead-end regions does not adversely affect standard GBFS because it never explores such regions. This is because the FF heuristic usually identifies that such unnecessary `open-new-stack` actions do not actually lead to a state which is closer to the goal. For example, in the example above, both S and S' will have the same heuristic value of 10. However, the heuristic does recognize that applying a `ship-order` action to S will lead to a state with a heuristic value of 9. A standard GBFS will therefore select a successor S_{ship} of S reached with a `ship-order` action. As the heuristic will continue to decrease when `ship-order` actions are applied to descendents of S_{ship}, all further node expansions will be from those in the search tree below S_{ship}. In avoiding such dead-end regions of the search space, GBFS makes the heuristic's weakness in such areas irrelevant.

Even when GBFS is used with deferred heuristic evaluation, the algorithm will still avoid such regions provided that it breaks ties in favour of the node which was placed on OPEN first. Recall that when using this technique, all successors of S will be placed in OPEN with a heuristic value of 10. If the first successor of S that is expanded is reached by applying a `ship-order` action in S, the resulting successor will have a heuristic value of 9 which will be assigned to all of its children. As such, the search will concentrate on the portion of the search tree under this node and bypass the dead-end region of the search entirely.

Even if S' is expanded before the other successors of S, this region will also be avoided if the search is using the tie-breaking scheme described above. This is because the children of S' will be assigned a heuristic value of S', which is 10. As S_{ship} will have the same evaluation of 10 but was put on OPEN earlier, it must be expanded before any successor of S'. Since the successors of S_{ship} will then be assigned a heuristic value of 9, the search will concentrate on the search tree under S_{ship} and bypass the dead-end region in which the heuristic is uninformative.

Note, that so as to be consistent with Fast Downward [5] and LAMA [14], this tie-breaking scheme was used in the experiments shown above.

A.5 Dead-End Regions and Heuristic Perturbation

Unfortunately, when heuristic perturbation is used, it can push the search into these dead-end regions. To see why, let us again consider state S as described above and assume a noise level of 1. Since deferred heuristic evaluation is being used, the states lead to by all the `ship-order` actions and the `open-new-stack` action will have the same non-noisy heuristic value of 10. If the `ship-order` actions are all assigned a noise value of 1, and the `open-new-stack` action is assigned a noise value of 0, then S' will be expanded. As described above, S' also has a non-noisy heuristic value of 10. This means that all of its children will end up being assigned a heuristic value of either 10 or 11. Since all of the other children of S have been assigned the heuristic of 11, this means that if even one of the children of S' achieved by using a `ship-order` action is assigned a noise value of 0, then it will be expanded. This child, denoted S'', cannot lead to a goal. However, due to the use of the `ship-order` action, the FF heuristic will instead indicate that progress has been made towards the goal. As such, it will have a non-noisy heuristic value of 9 and the search will continue to make progress into a dead-end region of the search space.

Even if the search does correctly expand a successor of S generated using a `ship-order` action, the search can still end up in a dead-end region. This is because `stacks-avail` will be 1 in the resulting state and 9 orders will still need to be shipped. As such, the situation is analogous to state S since the unnecessary use of action `open-new-stack` will again push the search into a dead-end region. As this same situation will occur on every step to the goal, the likelihood that the search enters a dead-end region of the space will increase. This can then cause the low coverage in this domain achieved with even low noise levels as seen in Table 1.

A.6 Later Representations of the Openstacks Domains

It is unclear why this same behaviour does not occur in the 2008 and 2011 versions of this problem. The encoding did change somewhat from 2006 to 2008 with the main result being that the same problem will have a shallower search tree when encoded using the 2008 representation when compared to the 2006 representation. Yet despite this difference, the dead-end regions described above still do occur in the 008 encoding. As such, this topic requires further study.

References

[1] Isabel Cenamor, Tomás de la Rosa, and Fernando Fernández. Learning Predictive Models to Configure Planning Portfolios. In *ICAPS Workshop on Planning and Learning*, 2013.

[2] Rina Dechter and Judea Pearl. Generalized Best-First Search Strategies and the Optimality of A*. *J. ACM*, 32(3):505–536, 1985.

[3] Ariel Felner, Sarit Kraus, and Richard E. Korf. KBFS: K-Best-First Search. *Annals of Mathematics and Artificial Intelligence*, 39(1-2):19–39, 2003.

[4] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2):100–107, 1968.

[5] Malte Helmert. The Fast Downward Planning System. *JAIR*, 26:191–246, 2006.

[6] Malte Helmert and Gabriele Röger. Fast Downward Stone Soup: A Baseline for Building Planner Portfolios. In *ICAPS-2011 Workshop on Planning and Learning*, pages 28–35, 2011.

[7] Jörg Hoffmann and Bernhard Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *JAIR*, 14:253–302, 2001.

[8] Tatsuya Imai and Akihiro Kishimoto. A Novel Technique for Avoiding Plateaus of Greedy Best-First Search in Satisficing Planning. In *AAAI*, 2011.

[9] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *ECML*, pages 282–293, 2006.

[10] Hootan Nakhost, Jörg Hoffmann, and Martin Müller. Resource-Constrained Planning: A Monte Carlo Random Walk Approach. In *ICAPS*, 2012.

[11] Hootan Nakhost and Martin Müller. Monte-Carlo Exploration for Deterministic Planning. In *IJCAI*, pages 1766–1771, 2009.

[12] Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1(3-4):193–204, 1970.

[13] Silvia Richter and Malte Helmert. Preferred Operators and Deferred Evaluation in Satisficing Planning. In *ICAPS*, 2009.

[14] Silvia Richter and Matthias Westphal. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *JAIR*, 39:127–177, 2010.

[15] Mark Roberts, Adele Howe, and Landon Flom. Learned models of performance for many planners. *ICAPS 2007 Workshop AI Planning and Learning*, pages 36–40, 2007.

[16] Jendrik Seipp, Manuel Braun, Johannes Garimort, and Malte Helmert. Learning Portfolios of Automatically Tuned Planners. In *ICAPS*, 2012.

[17] Richard Valenzano, Shahab Jabbari Arfaee, Jordan Thayer, Roni Stern, and Nathan R Sturtevant. Using Alternative Suboptimality Bounds in Heuristic Search. In *ICAPS*, 2013.

[18] Richard Valenzano, Hootan Nakhost, Martin Muller, Jonathan Schaeffer, and Nathan Sturtevant. Arvandherd: Parallel Planning with a Portfolio. In *ECAI*, pages 786–791. IOS Press, 2012.

[19] Richard Anthony Valenzano, Nathan R. Sturtevant, Jonathan Schaeffer, Karen Buro, and Akihiro Kishimoto. Simultaneously Searching with Multiple Settings: An Alternative to Parameter Tuning for Suboptimal Single-Agent Search Algorithms. In *ICAPS*, pages 177–184, 2010.