Finding Better Candidate Algorithms for Portfolio-Based Planners

Richard Valenzano and Jonathan Schaeffer
Department of Computing Science
University of Alberta
{valenzan,jonathan}@cs.ualberta.ca

Nathan Sturtevant
Department of Computer Science
University of Denver
sturtevant@cs.du.edu

Abstract

In order to construct a high-performance portfolio-based planner, a diverse set of candidate algorithms is needed. In our work, we are looking at the problem of constructing such candidate sets. Below we describe the `ArvandHerd` planner and use it to demonstrate the importance of selecting a candidate set that uses a variety of heuristic search algorithms. We then describe ways in which candidate algorithms can be constructed when there are atypical requirements on solution quality. Finally, we describe future work into building new candidate algorithms for the inclusion in portfolios.

Introduction

When developing a planning system for solving problems from a specific domain, a system designer may leverage prior knowledge about the domain. This approach has been quite successful in many domains such as robot motion planning (Stentz 1995) and DNA sequence alignment (McNaughton et al. 2002). However, by the very nature, such specialized systems are only suited for their target domain. As such, when faced with a new domain, system designers must start from scratch. Avoiding this time-consuming manual task helps to motivate the development of general purpose planning systems which can handle a wide variety of domains without prior domain knowledge or human assistance. Such systems not only act as a useful starting point for developing domain-specific planners for some new domain, but they also contribute towards one of the classic goals of artificial intelligence research: to develop general problem-solving systems that do not require human intervention.

Heuristic search is one popular way to build such general automated planners. This technique employs a *heuristic function* which estimates the distance that any state in the domain is from the goal. These functions are then used to guide a process through which candidate paths are iteratively constructed in an effort to find a complete solution paths.

The exact way in which candidate paths are built up and the ways in which heuristic information are used will vary from algorithm to algorithm. Due to this variety, the different heuristic search algorithms typically have different strengths and weaknesses with respect to how well they handle different types of domains and how they deal with error in the heuristic functions. This effect is exacerbated when dealing with general automated planners, as these systems use automatically generated heuristic functions which typically have different levels of accuracy on different domains (Hoffmann 2005). The result is that no single algorithm can be expected to dominate all competitors on all domains. This suggests the use of an *algorithm portfolio*. This technique involves using a set of algorithms independently on the same problem. By doing so, planners which use a portfolio can often combine the strengths of the included individual algorithms.

When selecting candidates for a portfolio-based planning system, diversity amongst the algorithms is of paramount importance. This means that the included algorithms should be complementary in their strengths and weaknesses. To do otherwise will result in a redundancy in the portfolio.

Yet many existing portfolio-based planners only consider candidate algorithms that are best-first search based. As such, our goal is to better understand how to best avoid redundancy when using multiple best-first search algorithms and to construct new candidate algorithms that do well to complement this popular existing approach. Doing so will require an examination of existing algorithms in terms of how they handle heuristic error, and the construction of new algorithms to complement them. For investigating the existing algorithms, we will construct artificial domains in which we can control the amount and type of heuristic error, so as to isolate the impact of the various error types on performance. This study will then help us develop new algorithms which will use stochasticity and random sampling to avoid overly committing to the strategy used by their unmodified counterparts. These new algorithms will then be evaluated on the aforementioned artificial domains, and we expect that they will be complementary to their unmodified versions and thereby offer new candidates for portfolio selection

This structure of the remainder of this document is as follows. We begin by briefly describing related work. Following that section, we will describe `ArvandHerd`, a multi-core portfolio-based planner that competed in the 2011 International Planning Competition. This work highlights the importance of ensuring diversity amongst the candidate algorithms, describes some inherent challenges in developing portfolio-based multi-core planners, and demonstrates that existing planners can be improved through the use of multiple configurations and stochasticity.

After our consideration of ArvandHerd, we look at the problem of developing algorithms for an arbitrary suboptimality requirement. In doing so, we introduce a functional notion of a suboptimality bound and show how existing ϵ-admissible algorithms can be modified so as to satisfy other types of bounds. This will provide us with many different candidates when developing algorithms for a given suboptimality and also gives system designers more choice in how they specify solution quality guarantees.

Finally, we conclude with a discussion of future work.

Related Work

In recent years, there has been an increased interest into building planners using portfolios. Much of this work involves learning how to best use an already constructed candidate set of algorithms. For example, the work of Roberts and Howe (2006) learns a decision tree offline based on training data so as to select an instance-specific portfolio for each new problem encountered. Fast Downward Stone Soup (FDSS) uses a similar approach in that the system includes an offline training phase which involves learning how an assignment of the available computational resources to different candidate algorithms (Helmert and Röger 2011). As our goal is to enrich the space of algorithms to select from, not to then most effectively use a candidate set, we consider these to be orthogonal to our own work.

Another notable paper is that of Seipp *et al.* (2012) in which they construct candidate sets by using an offline phase in which they use an automatic parameter tuner to learn a configuration of a planner for each of a set of training domains. The space of configurations being considered allows for the use of different heuristic functions, planning enhancements, and best-first search variants. However, in the next section we will show that even without tuning, strong portfolios can be built by increasing the diversity of the candidate algorithms and pairing best-first search with a random-walk based approach. This motivates our work in constructing new algorithms which specifically complement the existing ones.

ArvandHerd: Multi-Core Planning with a Portfolio

Ensuring that a portfolio contains candidate algorithms which can handle a variety of different domains is necessary for constructing high-performance portfolio-based planners. This intuitive idea thus became the driving principle in our development of ArvandHerd, a multi-core planner which won the multi-core sequential satisficing track of IPC 2011 (García-Olaya, Jiménez, and López 2011). The use of portfolios in this planner stands in contrast to past work on building parallel planning systems, such as PBNF (Burns et al. 2010) and HDA* (Kishimoto, Fukunaga, and Botea 2009), which has generally focused on parallelizing a single heuristic search algorithm. While these approaches have successfully improved run-time, satisficing planners that use these or similar techniques on shared memory machines should not be expected to solve many more problems than their single-core counterparts. This would be true for even perfect parallelizations of several state-of-the-art planners, which run exactly k times faster than their single-core counterparts when using k cores. Given a time limit T, the performance of such a k-core system can be simulated by running the single-core version of each planner for $k \cdot T$ time and counting any problem solved within this time limit as having been solved by the k-core parallelization in time T. This simulation indicates that even with such a speedup, coverage only increases slightly. The first three rows of Table 1 shows this simulated performance (in terms of coverage) for three different state-of-the-art planners on all 790 problems from the 2006, 2008, and 2011 IPC competitions. The three planners tested are LAMA-2008(Richter and Westphal 2010), LAMA-2011(Richter, Westphal, and Helmert 2011), and FDSS-1 (Helmert and Röger 2011). In all cases, the same pattern is clear: the use of additional cores results in almost no improvements in coverage.

Table 1: Performance of parallel planners.

Planner	Number of Cores			
	1	2	4	8
LAMA-2008 Sim	639.0	641.0	643.0	*NA*
LAMA-2011 Sim	721.0	724.0	726.0	727.0
FDSS-1 Sim	720.0	724.0	726.0	727.0
Parallel Arvand	660.4	668.0	677.8	679.6
ArvandHerd	*NA*	737.2	743.2	741.8

The issue with parallelizing a single-core planner is that the parallel version will most likely have the same weaknesses as its single-core counterpart. For example, all three of LAMA-2008, LAMA-2011, and FDSS-1 are memory heavy algorithms in which it is the available constraints on memory which are limiting the coverage. In such cases, any speedup seen through parallelizing the planners only causes memory to be exhausted more quickly. This behaviour is seen in the simulated LAMA-2011 parallelization, as the 8-core simulation ran out of memory on 52 problems. This means that regardless of how many more cores are used, at most 738 of the 790 problems can be solved using LAMA-2011 without an increase in memory.

Our approach to this problem was to use a carefully selected portfolio so as to combine the strengths of different planning techniques. The portfolio approach is particularly attractive due to its simplicity for implementation. This is because each member of the portfolio merely needs to be assigned to a different core. As such, the portfolio approach represents a simple alternative to the difficult process of parallelizing a single-core algorithm and it mostly avoids overhead from communication and synchronization.

In this end, we constructed ArvandHerd, a state-of-the-art parallel planner whose portfolio was selected specifically to avoid avoid the inherent limitations of parallelizing a single memory-heavy planning algorithm that were described above. Planners competing in this track were run on a 4-core machine with a maximum of 30 minutes of run-time and 6 GB of memory. In ArvandHerd, three cores were used to run a set of configurations of the linear-space random-walk-based planner Arvand, and the final processor was used to run the WA*-based LAMA-2008 planner.

Note, this section summarizes a post-competition analysis which showed that by using these two very different individual planners in a parallel portfolio, ArvandHerd is able to solve more IPC benchmark problems than several state-of-the-art planners, even if these state-of-the-art planners could be effectively parallelized (Valenzano et al. 2012). Below, we will also demonstrate that each of the Arvand and LAMA-2008 planners can be enhanced through the use of multiple configurations and restarts. While these techniques have been successfully applied in the satisfiability community, we demonstrate that they are similarly successful in planning.

The ArvandHerd Portfolio

LAMA-2008 (Richter and Westphal 2010), the winner of the sequential satisficing track of IPC 2008, was a state-of-the-art planner prior to IPC 2011 making it a natural selection for use in our portfolio. LAMA-2008 is WA*-based and can be memory-heavy. As such, although the ArvandHerd portfolio contains several configurations of LAMA-2008, it avoids having to partition the memory amongst the various configurations by only running a single LAMA-2008 configuration at a time. The additional LAMA-2008 configurations are only used if the first runs out of memory, in which case the planner *restarts* with another configuration.

The ArvandHerd portfolio also contains several configurations of Arvand (Nakhost and Müller 2009). This planner uses a random-walk-based search which makes it ideal for use alongside LAMA-2008 in a portfolio for several reasons. First, this approach is very different from WA* and it can solve some problems that the systematic search of WA* is unable to handle. Secondly, domains in which Arvand exhibits poor behaviour are often successfully tackled by WA*-based approaches. Finally, Arvand has low memory requirements, and so when it is run alongside LAMA-2008 in a shared-memory system, the majority of the memory can be assigned to LAMA-2008.

When ArvandHerd begins its search, separate threads are spawned to run different portfolio members. In a k-core machine setting, $k-1$ threads run a parallelization of Arvand while the remaining thread runs LAMA-2008.

We will not describe these planners in further detail here except to briefly note their complementary behaviour and how each can be improved through the use of multiple configurations. With respect to complementary behaviour, consider rows 1 and 4 of Table 1. The former shows the simulated performance of a parallelization of LAMA-2008 while the latter shows the actual average performance of a parallel version of Arvand over the same set of 790 problems. While Arvand outperforms LAMA-2008 in terms of its overall score, this is not necessarily the case when we consider the performance in a domain-by-domain fashion. Typically, domains in which single-core Arvand exhibits poor performance are also difficult for the multi-core versions. For example, neither the single-core nor the 8-core version of Arvand can solve even one of the 20 barman problems from IPC 2011. Similar behaviour is seen in the sokoban domain from IPC 2008, in which single-core Arvand solves an average of 4.4 of the 30 problems, while 8-core Arvand solves only 6.8. In contrast, LAMA-2008 is able to solve 15 of the 20 barman problems and 26 of the 30 sokoban problems. However, LAMA-2008 is able to solve only 19 of the 30 storage problems from IPC 2006 while Arvand is able to solve all 30. This further motivates the use of a portfolio containing these two planners.

Table 2: LAMA-2008 using restarts.

Search	Number of Restarts					
Type	0	1	2	4	8	16
GBFS	431.8	437.0	438.5	440.3	437.3	427.8
$w=7$	403.6	408.2	409.1	409.0	405.4	397.7
$w=1$	207.2	209.1	209.8	207.3	205.4	194.9

In our work, we also showed that by diversifying each of Arvand and LAMA-2008 through the introduction of stochasticity or multiple configurations we can improve the coverage of each individually. Here we only describe our findings in LAMA-2008 for which we showed that one powerful technique for improving coverage is *random operator ordering*. This technique involves randomly re-ordering the list of children of an expanded state before those new states are added to any open list. By restarting every so often and using random operator ordering, we can further extend the coverage of LAMA-2008. This can be seen in Table 2, which shows the coverage from LAMA-2008 when configured in 3 different ways (either to run greedy best-first search (GBFS) or WA* with weights of 7 or 1) when using different numbers of restarts. This is true of all configurations tested, including two others not shown. While this technique was known to work in simple single-agent search algorithms (Valenzano et al. 2010), our experiment shows that it can also benefit complete planning systems like LAMA-2008 even though it already uses multiple heuristics and multiple open lists so as to encourage diversity in its search.

Our experiments also suggests that if LAMA-2008 is set to restart not just with a new random seed but also with a different configuration, this leads to further coverage improvements. For example, when restarting 4 times such that each time uses a different one weight for WA*, the expected coverage is 448.4. This motivates the inclusion of several LAMA-2008 configurations in the portfolio.

ArvandHerd on IPC Benchmarks

The average performance of ArvandHerd when run 5 times on each of the 790 problems in the 2006, 2008, and 2011 is also shown in 1. The table shows that ArvandHerd 's coverage is significantly better than that of the Arvand parallelization and the perfectly linear parallelizations of LAMA-2011 and FDSS. ArvandHerd achieves its high coverage in the expected way, with Arvand and LAMA-2008 cancelling out each others weaknesses. For example, recall that Arvand is unable to solve even a single barman problem. With LAMA-2008 in the portfolio, 2-core `ArvandHerd` solves an average of 15.4 of the 20 problems (similar to the 16 solved by `LAMA-2008` when run on its own). Similarly, while `LAMA-2008` only solves 19 of 30 problems in `storage` (IPC 2006), 2-core `ArvandHerd` solves an average of 29.4 (similar to the 30 that `Arvand` solves when run on its own). In this way, `ArvandHerd` combines two planners in `LAMA-2008` and `Arvand` whose performance lag significantly behind `LAMA-2011` when used on their own to surpass even a perfectly linear parallelization of `LAMA-2011`.

Creating Algorithms for Alternative Suboptimality Bounds

While the algorithms used in `ArvandHerd` may find arbitrarily suboptimal solutions, some algorithms satisfy a *suboptimality bound*, which is a requirement on the cost of any solution that is set *a priori* of any problem-solving. By selecting a suboptimality bound, a user defines the set of solutions which are considered acceptable. For example, where C^* is the optimal solution cost for a task, the ϵ-admissible bound requires that any solution found must be from the set of solutions which have cost C satisfying $C \leq (1+\epsilon) \cdot C^*$. The first such algorithm to satisfy this bound is the commonly used Weighted A* (WA*) algorithm (Pohl 1970).

Now suppose that we wish to build a portfolio-based planner so as to satisfy a given suboptimality bound. A simple way to do this is to ensure that all candidates in the portfolio individually satisfy the bound. Clearly, this will guarantee that any solution found by the portfolio will also satisfy the suboptimality bound.

The questions that arises is then as follows: how can we construct a set of candidate algorithms for a given bound? In the case of ϵ-admissible bounds, there are plenty of algorithms to choose from including WA*, A$^*_\epsilon$ (Pearl and Kim 1982), Optimistic Search (Thayer and Ruml 2008), and EES (Thayer and Ruml 2011). Yet there also exists other types of suboptimality bounds. For example, suppose that the desired suboptimality guarantee is that any solution found has a cost C which satisfies the relation $C \leq C^* + \gamma$ for some $\gamma \geq 0$.

Building algorithms for such alternative bounds is the goal of a paper to appear at ICAPS 2013 (Valenzano et al. 2013). In that work, we show that many existing ϵ-admissible can be modified so as to apply to a large class of other suboptimality bounds. Doing so gives more choice into how suboptimality guarantees can be specified.

The contributions of our work are then as follows. First, we introduced a functional notion of a suboptimality bound so as to allow for the definition of alternative bounding paradigms. We then developed a theoretical framework which identifies how existing ϵ-admissible algorithms can be modified so as to satisfy alternative bounds. Finally, we demonstrated that the framework leads to practical algorithms that can effectively trade-off guaranteed solution quality for improved runtime when considered for additive bounds. In this section, we summarize these contributions.

Generalizing Suboptimality Bounds

Recall that the ϵ-admissible bound requires that the cost C of any solution returned must satisfy $C \leq (1+\epsilon) \cdot C^*$ for some given $\epsilon \geq 0$. We generalize this idea by allowing for an acceptable level of suboptimality to be defined using a function, $B : \mathbb{R} \rightarrow \mathbb{R}$. This *bounding function* is used to define the set of acceptable solutions as those with cost C for which $C \leq B(C^*)$. This yields the following definition:

Definition 1 *For a given bounding function B, an algorithm A will be said to* satisfy *B if on any problem, any solution returned by A will have a cost C for which $C \leq B(C^*)$.*

As an example of how this definition applies, let us again consider the ϵ-admissible requirement. The corresponding bounding function is $B_\epsilon(x) = (1+\epsilon) \cdot x$, and an algorithm is ϵ-admissible if and only if it satisfies B_ϵ. Similarly, an algorithm is optimal if and only if it satisfies the bounding function $B_{opt}(x) = x$. Other bounding functions of interest include $B_\gamma(x) = x + \gamma$ for some $\gamma \geq 0$, which corresponds to an *additive bound*, and $B(x) = C^* + \log C^*$, which allows for the amount of suboptimality to grow logarithmically with the optimal solution cost.

Notice that all of these bounding functions satisfy the relationship that $B(x) \geq x$ for all x. This is a necessary condition, as to do otherwise is to allow for bounding functions that require better than optimal solution quality. Any bounding function B satisfying this requirement will also be trivially satisfied by any optimal algorithm. However, selecting an optimal algorithm for a given B (where $B \neq B_{opt}$) defeats the purpose of even defining an acceptable level of suboptimality, which was to avoid the resource-intensive search typically required to find optimal solutions. The goal is therefore not only to find an algorithm that satisfies a given bounding function B, but to find an algorithm which satisfies B and can be expected to be faster than algorithms satisfying tighter bounds. The approach we take is similar to that of WA*: by allowing the algorithm to become greedier.

Building Algorithms for a Bounding Function

Having introduced a functional definition of a suboptimality bound, we now wish to construct algorithms for a giving bounding function. In our work, we investigate 4 different classes of existing algorithms and show that they each can be modified so as to be made to satisfy a large class of arbitrary bounding functions. These 4 classes are *anytime algorithms*, *best-first search algorithms*, *iterative deepening algorithms*, and *focal list based search*. In the interest of brevity, we only review the best-first search results here. Note, in the following we will let $h(n)$ denote the heuristic value associated with a node, $g(n)$ will denote the cost of the path found to n, and h is said to be *admissible* if for all nodes n, $h(n) \leq h^*(n)$ where $h^*(n)$ is the true distance from n to the nearest goal.

We define best-first search as a generalization of A* and Djikstra's algorithm whereby some arbitrary evaluation function Φ is used to guide the search. For example, the A* algorithm (Hart, Nilsson, and Raphael 1968) is guided by the evaluation function $\Phi(n) = f(n) = g(n) + h(n)$ where h is admissible, and Dijkstra's algorithm (Dijkstra 1959) is guided by $\Phi(n) = g(n)$. For convenience, we use the notation BFS$^{\Phi}$ to denote a BFS instance which is using the evaluation function Φ.

Our main theorem for best-first search is the following:

Theorem 0.1 *Given a bounding function B, BFS^{Φ} will satisfy B if Φ is a function that satisfies the following:*

1. *$\forall$ node n on some optimal path, $\Phi(n) \leq B(g(n)+h^*(n))$*
2. *$\forall$ goal node n_g, $g(n_g) \leq \Phi(n_g)$*

Notice that Theorem 0.1 does not suggest a particular evaluation function for satisfying a given bounding functions but instead defines a space of evaluation functions that will suffice. This theorem therefore identifies a space of candidate algorithms for inclusion in a portfolio-based system for satisfying B.

For a certain class of bounding functions, Theorem 0.1 can also be used to construct a specific algorithm for satisfying the desired bound. This result is inspired by WA* (Pohl 1970). Recall that it uses the evaluation function $f_\epsilon(n) = g(n) + (1+\epsilon)\cdot h(n)$ (*ie.* WA* is BFS$^{f_\epsilon}$) and is ϵ-admissible if h is admissible. Also notice that $f_\epsilon(n) = g(n)+B_\epsilon(h(n))$ where B_ϵ is the bounding function corresponding to ϵ-admissibility. In practice, WA* often solves problems faster than A*. This is because f_ϵ increases the importance of h relative to g, and therefore allows WA* to search more greedily on h than does A*. This suggests the use of the evaluation function $\Phi_B(n) = g(n) + B(h(n))$ for satisfying a given bounding function B since it similarly puts additional emphasis on h. The following corollary shows that this approach can be used on a large class of bounding functions:

Corollary 0.2 *Given bounding function B such that for all x, y, $B(x+y) \geq B(x)+y$, if $\Phi_B(n) = g(n) + B(h(n))$, then BFS$^{\Phi_B}$ with satisfy B.*

Algorithms with Additive Bounds

So as to demonstrate the utility of the theoretical framework just described, we used it to construct algorithms for the additive bounding function $B_\gamma(x) = x + \gamma$ for some $\gamma \geq 0$. This bounding function represents a special case in which the evaluation function offered in Corollary 0.2 will result in an algorithm which behaves identically to A*. Since the whole reason for allowing for suboptimal algorithms in the first case was to achieve better performance than A*, using this evaluation function is inadequate. Instead, we will use the following evaluation function:

$$F_\gamma(n) = g(n) + h(n) + \frac{\min(h(n), h(n_i))}{h(n_i)} \cdot \gamma$$

where n_i is the initial node. By Theorem 0.1, BFS$^{F_\gamma}$ can be shown to satisfy B_γ.

Intuitively, F_γ penalizes nodes inversely with how much *heuristic progress* has been made (as measured by $h(n)/h(n_i)$), with the maximum possible penalty being γ. Notice that if $h(n_i)$ is the largest heuristic value seen during the search, then the 'min' can be removed from F_γ which then becomes equivalent to ϵ-admissible evaluation function $f_\epsilon(n) = g(n) + (1+\epsilon)\cdot h(n)$ in which $\epsilon = (1 + \gamma/h(n_i))$. This suggests that BFS$^{F_\gamma}$ can be expected to behave similarly to WA*, though while achieving a completely different kind of suboptimality bound.

We experimented with BFS$^{F_\gamma}$ by implementing it into the Fast Downward (Helmert 2006) framework. The experiments were performed on the 280 problems from the optimal track of IPC 2011, with the planner being given maximums of 30 minutes and 4 GB per problem. The admissible heuristic used is LM-Cut (Helmert and Domshlak 2009).

The general behaviour seen was as follows: as suboptimality is allowed to increase, so too does the coverage. When $\gamma = 0$, BFS$^{F_\gamma}$ is the same as A* and it is able to solve 132 of the problems. With even a slight increase to $\gamma = 1$, an total of 145 problems are solved — representing an increase of almost 10%. The coverage then increases to 163, 204, and 218 problems solved for γ values of 10, 100, and 1000, respectively. This is consistent with the behaviour of WA*, which also benefits from the additional greediness allowed with a looser bound.

In (Valenzano et al. 2013), we also show similar gains when using iterative deepening so as to satisfy additive bounds in large combinatorial puzzles, and focal list based algorithms so as to satisfy additive bounds in domains with non-uniform action costs.

Future Work

The work on bounded portfolios in the previous section represents several opportunities for future work. While we have shown that we can modify the different classes of ϵ-admissible algorithms so as to satisfy other bounds, we have yet to experiment with them together in a portfolio setting. As the different algorithm classes have their own specialities — with best-first search being most effective on uniform cost domains, focal list based search being most effective on non-uniform cost domains, and iterative deepening being most effective on large combinatorial domains with few cycles — we expect that they will do well when used together. However, this remains to be shown experimentally.

Another aspect of these new algorithms that needs investigation is under which conditions they can be expected to effectively trade-off solution quality for search speed. There has been some work on showing that WA* can be expected to have polynomial performance under certain types of heuristic error (Chenoweth and Davis 1992). Our intent is to extend this result to other algorithms of the type constructed by Corollary 0.2 so as to determine when these new algorithms can similarly be expected to be effective. By finding the error types under which these different algorithms should perform well, we can construct sets of candidate algorithms designed specifically so as to handle a wide range of heuristic error. This will be useful not only for bounded suboptimal search but also to general automated planners which also need to handle a variety of heuristic error types.

Currently, the most popular heuristic search algorithm for use in satisficing planning is greedy best-first search (GBFS). While including algorithms for handling other types of heuristic error has been described above, we are also attempting to construct alternative forms of this algorithm that do not fail in similar situations. GBFS often finds solutions quickly in practice due to its greediness, but that same greediness means that it can easily be lead astray by inaccurate heuristic information. In an effort to consider algorithms that can complement GBFS in a portfolio we have looked at constructing search procedures which does not trust the heuristic information too much. This means that occasionally these new algorithms should expand a node with a higher h-cost than one with a lower h-cost.

We have had some early success in doing so using *heuristic perturbation*. Given a heuristic h, GBFS with heuristic perturbation (GBFS-HP) performs a GBFS search using the heuristic h' defined as $h'(n) = h(n) + r(n)$, where $r(n)$ is a random value uniformly selected from the range $[-\tau, \tau]$. $\tau \geq 0$ is a user set algorithm parameter. The value of $r(n)$ is selected immediately prior to n being added to OPEN, and so it can result in a different ordering of the nodes in the open list than would occur when using heuristic h.

GBFS-HP was implemented into `LAMA-2008` and tested against standard GBFS using only a single heuristic and no preferred operators. On the 560 problems from the IPC 2006 and IPC 2008, standard GBFS solved 339 problems. With $\tau = 5$ and $\tau = 10$, GBFS-HP was able to solve an average of 360.2 and 382.6 problems when run 5 times per problem. Moreover, its behaviour is very different than that of GBFS with there being 8 domains in which GBFS-HP with $\tau = 10$ having solved 10% more problems than standard GBFS and there being 2 domains with the opposite happening.

While GBFS-HP appears to have much potential for inclusion alongside GBFS in a portfolio, more investigation into this algorithm is needed. A better understanding of this algorithm will be performed by testing it on artificial search domains in which the heuristic error can be controlled and modified.

References

Burns, E.; Ruml, W.; Lemons, S.; and Zhou, R. 2010. Best-First Heuristic Search for Multicore Machines. *JAIR* 39:689–743.

Chenoweth, S. V., and Davis, H. W. 1992. New approaches for understanding the asymptotic complexity of a tree searching. *Ann. Math. Artif. Intell.* 5(2-4):133–162.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

García-Olaya, A.; Jiménez, S.; and López, C. L. 2011. IPC 2011 Deterministic Track. http://ipc.icaps-conference.org.

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

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS*.

Helmert, M., and Röger, G. 2011. Fast Downward Stone Soup: A Baseline for Building Planner Portfolios. In *ICAPS-2011 Workshop on Planning and Learning*, 28–35.

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

Hoffmann, J. 2005. Where 'ignoring delete lists' works: Local search topology in planning benchmarks. *J. Artif. Intell. Res. (JAIR)* 24:685–758.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2009. Scalable, Parallel Best-First Search for Optimal Sequential Planning. In *ICAPS*.

McNaughton, M.; Lu, P.; Schaeffer, J.; and Szafron, D. 2002. Memory-efficient a* heuristics for multiple sequence alignment. In *AAAI/IAAI*, 737–743.

Nakhost, H., and Müller, M. 2009. Monte-Carlo Exploration for Deterministic Planning. In *IJCAI*, 1766–1771.

Pearl, J., and Kim, J. H. 1982. Studies in Semi-Admissible Heuristics. *IEEE Trans. on Pattern Recognition and Machine Intelligence* 4(4):392–399.

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

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

Richter, S.; Westphal, M.; and Helmert, M. 2011. LAMA 2008 and 2011. In *IPC 2011 Deterministic Track*, 117–124.

Roberts, M., and Howe, A. 2006. Directing a portfolio with learning. In *AAAI 2006 Workshop on Learning for Search*, 129–135.

Seipp, J.; Braun, M.; Garimort, J.; and Helmert, M. 2012. Learning portfolios of automatically tuned planners. In *ICAPS*.

Stentz, A. 1995. The focussed d* algorithm for real-time replanning. In *IJCAI*, 1652–1659.

Thayer, J. T., and Ruml, W. 2008. Faster than Weighted A*: An Optimistic Approach to Bounded Suboptimal Search. In *ICAPS*, 355–362.

Thayer, J. T., and Ruml, W. 2011. Bounded Suboptimal Search: A Direct Approach Using Inadmissible Estimates. In *IJCAI*, 674–679.

Valenzano, R.; Sturtevant, N.; Schaeffer, J.; Buro, K.; and Kishimoto, A. 2010. Simultaneously Searching with Multiple Settings: An Alternative to Parameter Tuning for Suboptimal Single-Agent Search Algorithms. In *ICAPS*, 177–184.

Valenzano, R. A.; Nakhost, H.; Müller, M.; Schaeffer, J.; and Sturtevant, N. R. 2012. Arvandherd: Parallel planning with a portfolio. In *ECAI*, 786–791.

Valenzano, R. A.; Arfaee, S. J.; Thayer, J. T.; Stern, R.; and Sturtevant, N. 2013. Using alternative suboptimality bounds in heuristic search. In *ICAPS*.