Artificial Intelligence 267 (2019) 39–57

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Probably bounded suboptimal heuristic search

Roni Stern [a,*], Gal Dreiman [a], Richard Valenzano [b]

[a] Ben Gurion University of the Negev, Israel
[b] University of Toronto, Canada

ARTICLE INFO

Article history:
Received 10 March 2017
Received in revised form 15 June 2018
Accepted 8 August 2018
Available online 25 October 2018

Keywords:
Artificial intelligence
Heuristic search

ABSTRACT

Finding an optimal solution to a search problem is often desirable, but can be too difficult in many cases. A common approach in such cases is to try to find a solution whose suboptimality is bounded, where a parameter ϵ defines how far from optimal a solution can be while still being acceptable. A scarcely studied alternative is to try to find a solution that is *probably optimal*, where a parameter δ defines the confidence required in the solution's optimality. This paper explores this option and introduces the concept of a *probably bounded-suboptimal* search (pBS search) algorithm. Such a search algorithm accepts two parameters, ϵ and δ, and outputs a solution that with probability at least $1-\delta$ costs at most $1+\epsilon$ times the optimal solution. A general algorithmic framework for pBS search algorithms is proposed. Several instances of this framework are described and analyzed theoretically and experimentally on a range of search domains. Results show that pBS search algorithms are often faster than a state-of-the-art bounded-suboptimal search algorithm. This shows in practice that finding solutions that satisfy a given suboptimality bound with high probability can be done faster than finding solutions that satisfy the same suboptimality bound with certainty.

1. Introduction

Consider a search problem in which we must find a path in a state space from a given initial state to a goal state. Given enough memory and running time, standard heuristic search algorithms like the A* algorithm [1] can solve a given search problem *optimally*, i.e., find the lowest-cost path from the initial state to a goal. However, it is often the case that there is not enough memory or runtime to find an optimal solution. For such cases, a range of search algorithms have been proposed that return *suboptimal* solutions [2–5]. In particular, *bounded-suboptimal search algorithms* are algorithms that are guaranteed to return a solution whose cost is at most $(1+\epsilon)$ times the optimal solution. Ideal bounded-suboptimal algorithms introduce a natural tradeoff between solution quality and search runtime: when ϵ is high, solutions are returned quickly but can have poorer quality, while when ϵ is low solutions are harder to find but usually have higher quality (i.e., lower cost).

Bounded-suboptimal search algorithms are very strict, in the sense that the cost of the solution they return must be at most $1+\epsilon$ times the cost of an optimal solution. Current techniques to achieve this guarantee rely on *admissible* heuristics – heuristics that are a lower bound on the optimal solution cost. Such heuristics are often inaccurate, resulting in increased search runtime.

* Corresponding author.
E-mail addresses: sternron@post.bgu.ac.il (R. Stern), gal.dreiman@gmail.com (G. Dreiman), rvalenzano@cs.toronto.edu (R. Valenzano).

https://doi.org/10.1016/j.artint.2018.08.005
0004-3702/

In this paper, we propose an alternative type of solution quality guarantee. Search algorithms with this quality guarantee return solutions that are optimal with high probability. Instead of controlling the suboptimality of the returned solution (i.e., ϵ), these algorithms accept a parameter δ that controls the *confidence* in which the returned solution is optimal. δ allows a similar tradeoff of solution quality and runtime, where increasing δ is expected to decrease the search effort at the cost of increasing the likelihood that a suboptimal solution will be returned.

Both types of solution quality guarantees can be combined, through the novel notion of a *probably bounded-suboptimal search* (pBS search). A pBS search algorithm is given two parameters, ϵ and δ, and is required to return a solution that is at most $1+\epsilon$ times the optimal solution, with probability higher than $1-\delta$. We call $1+\epsilon$ the *desired suboptimality*, and $1-\delta$ the *required confidence*. Introducing and defining the concept of a pBS search is the **first main contribution of this work**.

In many domains the solution found by current bounded-suboptimal search algorithms has a much lower (i.e., better) suboptimality in practice than the suboptimality guaranteed by the bound. Since there is usually a tradeoff between runtime and solution quality, this means the user of the search algorithm paid in runtime more than it was needed for the desired solution quality. A key benefit of the novel form of bounded suboptimality we propose is that it provides users of search algorithms with more control over the time-quality tradeoff. We observed this experimentally: by using some of the pBS algorithms we propose it is often possible to obtain a solution with the desired suboptimality significantly faster by only slightly relaxing the required confidence from 1.0 to 0.9.

The **second contribution** is a general framework for developing pBS algorithms called Psf. Psf has two main building blocks: a solution generator and a stopping condition. The solution generator can be any algorithm that produces a sequence of solutions. The stopping condition is responsible for identifying when the current solution is sufficient, in the sense that it has the desired suboptimality with the required confidence. We propose three such stopping conditions – Absolute, h-ratio, and Open-based – and prove that using these conditions results in a pBS search algorithm. For the Absolute and h-ratio conditions we also propose a solution generator that is specifically designed to satisfy these stopping conditions quickly. These stopping conditions and new solution generator are the **third contribution** of this work.

Finally, we evaluate the different instances of Psf experimentally on four search domains: the Pancakes puzzle, Dockyard robot, Vacuum cleaner, and grid-based pathfinding. The experiments on these diverse set of domains show that varying both ϵ and δ offers a flexible control over the solution quality versus runtime tradeoff: in general, increasing either ϵ or δ results in smaller running time and lower solution quality. In particular, setting $\delta > 0$ indeed allows us to find solutions faster.

Some of the material in this work was previously published in the proceedings of the Symposium on Combinatorial Search (SoCS) [6,7]. This paper summarizes and goes well beyond these two conference papers. In particular, it extends these prior works by:

- The experimental evaluation in these prior conference publications considered only one domain: the 15-puzzle. This work significantly extends that evaluation through the implementation and evaluation of all Psf instances on four additional domains (see Section 5).
- The theoretical basis of pBS search is properly defined and analyzed. This includes several corrections to the original work (see Appendix B).
- For the Absolute and h-ratio stopping conditions, we propose a solution generator that is specifically designed for them, which is based on recent bounded-cost search algorithms (see Section 6).

In addition, in previous work [6,7], this line of research was referred to as Probably Approximately Correct (PAC) search, since it was inspired by the notion of PAC learning from the theoretical machine learning literature [8]. Given that there are significant differences between the concept of PAC and the solution quality guarantees considered in this paper, we have changed the name to *probably bounded-suboptimal search* to avoid confusion. A comparison of these related concepts can be found in Section 8.

2. Preliminaries and background

A *graph search problem* or a *search problem* is defined by a graph G, a source vertex $s \in V$, and a non-empty set of target vertices $T \subseteq V$. The edges in G are associated with a non-negative cost, denoted by $c(e)$, and the cost of a path p in G, denoted $c(p)$, is the sum of costs of its constituent edges. A solution to a search problem P is a path in G from s to a vertex in T. A solution is called *optimal* if there is no other solution that has a lower cost. Let Opt_P denote the cost of the optimal solution to P. The *suboptimality* of a solution is the ratio between its cost and Opt_P. Hence, the suboptimality of an optimal solution is 1. Note that alternative forms of suboptimality have also been introduced [9] and the concepts in this paper can easily be extended to them. For clarity, we focus in this paper on the aforementioned definition of suboptimality.

A *search algorithm* is a procedure that accepts a search problem and tries to return a solution to it. Note that in some cases the underlying graph G is not given to the search algorithm explicitly, e.g., because it is too large to fit in memory. Instead, the search algorithm can be given a source vertex s and a set of state transition operators, which implicitly define G as the set of all states reachable by applying sequences of state transition operators to s.

2.1. Properties of search algorithms

Let $cost(A, P)$ denote the cost of the solution returned by algorithm A when given problem P. A search algorithm A is called an *optimal search algorithm* if for every search problem P it holds that $cost(A, P) = \text{OPT}_P$. A search algorithm A is called a *bounded-suboptimal search algorithm* if it accepts a parameter ϵ and for every problem P it holds that $cost(A, P) \leq (1 + \epsilon) \cdot \text{OPT}_P$, i.e., the suboptimality of the solutions it returns is at most $1 + \epsilon$. A* [1], IDA* [10], and RBFS [11] are examples of optimal search algorithms, while Weighted A* [2], A*$_\epsilon$ [3], Explicit Estimation Search [4], and Dynamic Potential Search [5] are examples of bounded-suboptimal search algorithms.

Two additional types of search algorithms that we use in this paper are *anytime search* algorithms and *bounded-cost search* algorithms. Anytime search algorithms continue to search for better solutions after an initial solution is found, if given more running time. Prominent examples of anytime search algorithms are Anytime Weighted A* [12], Beam-Stack Search [13], Anytime Window A* [14], and Anytime Potential Search (APTS) [15], also known as Anytime Non-parametric A* (ANA*). A search algorithm is a *bounded-cost search (BCS) algorithm* if it accepts a parameter B and it returns a solution of cost at most B, if such a solution exist. Potential Search (PTS) [15] and Bounded-cost Explicit Estimation Search (BEES) [16] are examples of BCS algorithms.

Note that our focus in this work is on graph search problems where finding a solution with a smaller cost is desired. Search algorithms have been applied to other settings also, e.g., where the goal is to find a solution with maximal cost [17]. We expect that extending our results to such settings will not be difficult.

2.2. Best-first search

Many heuristic search algorithm fit into the general framework of *best-first search*. Best-first search (BFS) is an iterative algorithm that maintains a list of nodes called OPEN. On every iteration, this algorithm chooses a single node from OPEN to *expand*, where expanding a node means generating its children and inserting them to OPEN. Initially OPEN contains only the initial state s_i. BFS algorithms also maintain another list of nodes called CLOSED, which contains all the previously expanded nodes. CLOSED is used to avoid generating states multiple times if not needed.

Best-first search algorithms differ in how they choose which node to expand from OPEN. For example, the A* algorithm [1] chooses to expand the node in OPEN that has the minimal $g + h$ value, where the g value of a node n, denoted $g(n)$, is the lowest cost path found so far from s_i to n, and $h(n)$ is a heuristic estimate of the cost from n to a goal node. If the heuristic function is *admissible* (i.e., is always a lower bound) then A* is guaranteed to find an optimal (lowest-cost) solution [1].

3. Probably bounded suboptimal (pBS) search

In this section, we define the notion of *probably bounded-suboptimal search*. Let $\mathcal{P}$ be a set of search problems and let D be a distribution over them. P_D is defined as a random variable drawn from $\mathcal{P}$ according to distribution D. Following this notation, $cost(A, P_D)$ is the random variable of the cost of the solution returned by search algorithm A when given a search problem drawn from $\mathcal{P}$ according to distribution D. Similarly, OPT_{P_D} is the random variable of the optimal solution cost of problems drawn from $\mathcal{P}$ according to D.

Definition 1 *(pBS search algorithm).* A search algorithm is a pBS search algorithm w.r.t. $\mathcal{P}$ and D iff it accepts as input $\epsilon > 0$ and $1 > \delta > 0$ and with probability at least $1 - \delta$ it outputs a solution with suboptimality of at most $1 + \epsilon$, i.e.,

$$Pr\Big(cost(A, P_D) \leq (1 + \epsilon) \cdot \text{OPT}_{P_D}\Big) \geq 1 - \delta \tag{1}$$

We refer to $1 + \epsilon$ as the *desired suboptimality* and refer to $1 - \delta$ as the *required confidence*. Thus, a pBS search algorithm is a search algorithm that returns a solution that has the desired suboptimality with the required confidence.

Classical search algorithms can be viewed as special cases of a pBS search algorithm. Optimal search algorithms are pBS search algorithms, since if $cost(A, P) = \text{OPT}_{P_D}$ then clearly Equation (1) also holds for A for any non-negative ϵ and δ. Similarly, any bounded-suboptimal search algorithm is also a pBS search algorithm, since if $cost(A, P) \leq (1 + \epsilon) \cdot \text{OPT}_{P_D}$ then clearly Equation (1) holds also for any non-negative δ.

However, both classes of algorithms ignore the value of δ, and thus miss out on potential opportunities for speeding up the search. Next, we propose a general framework for a pBS search algorithm that considers both ϵ and δ.

3.1. A framework for pBS search algorithms

The pBS search framework we propose, abbreviated as PSF, has two major building blocks: a *solution generator* and a *stopping condition*. The role of the solution generator is to find solutions, such that each solution has a lower cost than the previous one. The role of the stopping condition is to decide when the best solution found so far – the incumbent solution – has the desired suboptimality with the required confidence.

Algorithm 1: pBS search algorithm framework (PSF).

```
   Input: 1 + ϵ, the desired suboptimality
   Input: 1 − δ, the required confidence
 1 U ← ∞;
 2 Incumbent ← None
 3 while Improving U is possible do
 4 |   NewSolution ← GenerateSolution(U)
 5 |   if NewSolution is empty then
 6 |   |   return Incumbent
 7 |   else
 8 |   |   Incumbent ← NewSolution
 9 |   |   U ← NewSolution.Cost
10 |   |   if ShouldStop(U) then return Incumbent
11 |   end
12 end
```

PSF uses these building blocks as follows (see pseudo code in Algorithm 1). First, the solution generator (line 4 in Algorithm 1) is run to find an initial solution. If it successfully finds a solution then that solution and its cost are stored in variables *Incumbent* and U, respectively. Following, the stopping condition (`ShouldStop`, line 10) checks if U has the desired suboptimality with the required confidence.[1] If so then *Incumbent* is returned. Otherwise, another iteration begins in which the solution generator finds a new solution, updating *Incumbent* and U, and then the stopping condition is invoked to check if U has the desired suboptimality with the required confidence (lines 3–11). This iterative process continues until an incumbent solution that satisfied the stopping condition is found, or until the solution generator cannot find more solutions. Note that the solution generator accepts U as a parameter, as it seeks to find a solution whose cost is lower than *Incumbent*'s cost. If no such solution exists, *Incumbent* is returned. Returning *Incumbent* in this case is safe if the solution generator is complete, since if it cannot find a solution with cost lower than U then *Incumbent* is optimal.

The key question when implementing PSF is how to implement the solution generator and the stopping condition. *Anytime* search algorithms are especially suitable to serve as solution generators: given enough time they return a sequence of solutions such that each subsequent solution is better than its predecessor. Moreover, it is straightforward to modify many existing anytime search algorithms so that they accept the cost of the incumbent solution (U) and prune parts of the search space that cannot result in improving the incumbent solution. For example, if an anytime search algorithm uses an admissible heuristic then it can prune nodes with $g + h \geq U$.

Some anytime algorithms are guaranteed to eventually find an optimal solution. If PSF uses such an anytime algorithm as a solution generator then there exists a stopping condition for which PSF is sound and complete, i.e., for any value of ϵ and δ it will find a solution that has the desired suboptimality with the required confidence. An example of such a stopping condition is a stopping condition that is never satisfied, i.e., never halts the search. In such a case, the solution generator will eventually find an optimal solution, which satisfies any desired suboptimality requirements. For all the discussions below, we assume that the solution generator being used is one that converges to an optimal solution.

4. pBS stopping conditions

Clearly, using a stopping condition that never halts is inefficient. In this section, we propose better stopping conditions that can halt the search earlier and still maintain the pBS search guarantees (Definition 1). Such stopping conditions are referred to as *pBS conditions*.

Definition 2 *(pBS condition).* A pBS condition is a stopping condition for a PSF algorithm that guarantees that any instance of PSF using this stopping condition and a complete solution generator will be a pBS search algorithm.

The notion of pBS conditions and pBS search are all defined with respect to the set of problems $\mathcal{P}$ and the distribution D over them. To develop effective pBS conditions, we assume a pre-processing stage in which we are given a set of search problems, each sampled independently from the same distribution (D) (i.e., they were sampled in an i.i.d. manner) and we have sufficient resources to solve them optimally. We refer to this given set of search problems as the *training set*. Having a representative training set and being able to solve optimally the problems in it is certainly not always possible. However, these are reasonable assumptions in many realistic scenarios. See the discussion on this in Section 7.

4.1. The ABSOLUTE condition

Having a sample of problems and their optimal solutions allows us to approximate the *cumulative distribution function* (CDF) of the random variable OPT_{P_D}. This can be done, for example, by counting the number of problems with an optimal

[1] As we later discuss, creating such a stopping condition is non-trivial, and large parts of this paper are devoted to proposing such stopping conditions.

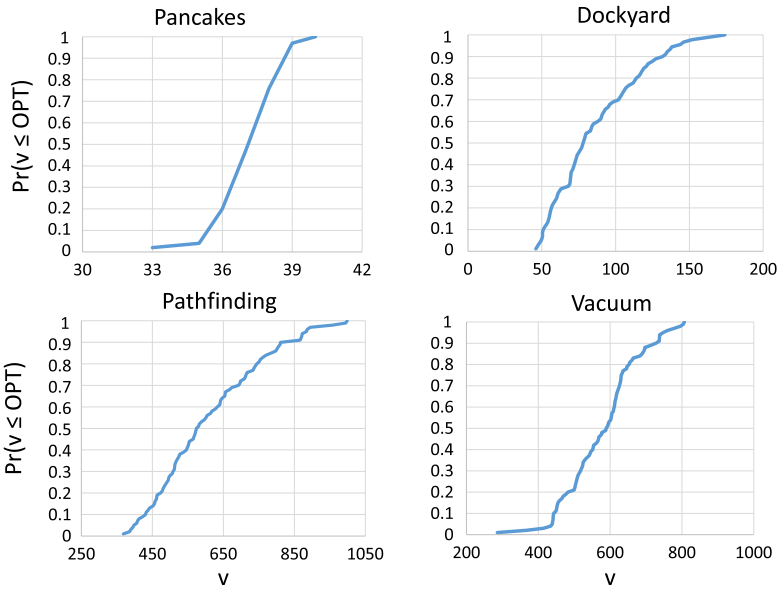

Fig. 1. The distribution of $F(v)$ for the 4 domains used in our experimental results. The x axis shows optimal solution costs, denoted by v, and the y axis shows the $F(v)$ value.

solution higher than v for a range of v values. Other statistically valid curve fitting techniques can also be used. Let $F(v)$ denote the resulting CDF. How close $F(v)$ is to the actual CDF of OPT_{P_D} depends on the size of the training set and the approximation technique used. For simplicity, we assume throughout this paper that the $F(v)$ is the actual CDF of OPT_{P_D}, i.e., we assume that $F(v) = Pr(v \leq \text{OPT}_{P_D})$. Future work will study how the number of problems used to create this CDF affects the resulting inaccuracy, and perhaps modify δ to incorporate this error.

Fig. 1 shows the CDFs we used in our experiments, which were generated using the simple problem-counting method mentioned above. The x axis shows the possible values of OPT_{P_D} and the y axis shows the cumulative distribution, i.e., for a given x value it shows the value of $Pr(x \leq \text{OPT}_{P_D})$. See Section 5 for a description of the different domains.

For a given CDF $F(v)$, we define a threshold value $T(\epsilon, \delta)$ as the maximal number v for which $F(\frac{v}{1+\epsilon})$ is equal to or larger than $1 - \delta$. Formally,

$$T(\epsilon, \delta) = \underset{v \geq 0}{\text{argmax}} \left(F\left(\frac{v}{1+\epsilon}\right) \geq 1 - \delta \right)$$

$T(\epsilon, \delta)$ is key in the first pBS condition we introduce, which we call the Absolute condition. According to this condition, the search halts if $U \leq T(\epsilon, \delta)$.

Theorem 1. *The* Absolute *stopping condition is a pBS condition.*

Proof. Let P be a problem drawn from $\mathcal{P}$ according to distribution D, and let A be an instance of Psf that uses the Absolute condition. When A halts on P it either has found an optimal solution (line 6 in Algorithm 1) or it halted because Absolute was satisfied (line 10). In the worst case, A always halts due to the latter reason. In this case, $cost(A, P) \leq T(\epsilon, \delta)$. This means that

$$F\left(\frac{cost(A, P)}{1+\epsilon}\right) \geq 1 - \delta \tag{2}$$

$$Pr(\frac{cost(A, P)}{1+\epsilon} \leq \text{OPT}_{P_D}) \geq 1 - \delta \tag{3}$$

$$Pr(cost(A, P) \leq (1+\epsilon) \cdot \text{OPT}_{P_D}) \geq 1 - \delta \tag{4}$$

Since P is sampled from P_D, it follows that its optimal solution OPT_P is sampled from the distribution of optimal solutions OPT_{P_D}, and thus

$$Pr(cost(A, P) \leq (1+\epsilon) \cdot \text{OPT}_P) \geq 1 - \delta \quad \square \tag{5}$$

The Absolute condition has several attractive properties. First, it is very easy to implement: simply halt when the incumbent solution is below $T(\epsilon, \delta)$. Thus, this condition is easy to integrate in existing search algorithms. Second, it requires

virtually no additional memory or running time, since the threshold value $T(\epsilon, \delta)$ does not depend on the problem at hand (P) and can be computed once for all problems in this domain.

The ABSOLUTE condition does not consider any information about the search problem that is currently being solved. To illustrate why this can be a wasteful, consider the following example.

Example 1. Assume that we have an admissible heuristic function h, and we are trying to solve a search problem P with start state s_i such that $h(s_i) = 40$. Since h is admissible, we know that OPT_P cannot be below 40. Thus, if we find an incumbent solution of cost 40 we can immediately halt, regardless of the value of δ. However, if our training set contained even one state with an optimal solution of 35 then we will have $F(40) > 0$ and thus for some values of δ the ABSOLUTE condition will not halt the search. Moreover, assume that we are using a heuristic that almost always underestimates the optimal solution by 5. In such a case, if we happen to find a solution of cost 44 then it is most likely optimal and we can halt. Again, the ABSOLUTE condition is agnostic to this, and will not exploit this information.

4.2. Heuristic-aware pBS conditions

We now present two complementary solutions to the issues identified by the example above.

4.2.1. Bounding the optimal solution

Many search algorithms maintain a lower bound on the optimal solution. For example, any search algorithm that maintains all the generated nodes in an open list, such as A* [1], WA* [2], ARA* [18], and APTS [5], can use $f_{min} = \min_{n \in \text{OPEN}}(g(n) + h(n))$ as a lower bound on the optimal cost, if h is admissible. The value of f_{min} may increase and decrease (e.g., when using inconsistent heuristics [19]) as nodes are expanded and generated. So, to obtain the tightest lower bound some search algorithms maintain the maximal f_{min} observed so far. We denote this value as MAX_{fmin}.

Clearly, if the incumbent solution is equal to or smaller then $(1+\epsilon)\cdot \text{MAX}_{fmin}$, then it has the desired suboptimality with complete certainty (i.e., even for $\delta = 0$). This stopping condition, which we refer to here as the MAX_{fmin} condition, is well-known in the literature on anytime search algorithms [12] and bounded-suboptimal search algorithms [4,5,3]. Importantly, the MAX_{fmin} condition can be used together with any other pBS condition by halting the search if either condition is satisfied. We thus implemented this additional stopping condition with all of our pBS conditions. Observe that the MAX_{fmin} condition resolves the first problem in Example 1, where $h(s_i) = 40$ and the cost of the incumbent solution is also 40, since if $h(s_i) = 40$ then $\text{MAX}_{fmin} \geq 40$.

4.2.2. h-RATIO condition

The following pBS condition, called the h-RATIO condition, aims to resolve the second problem in Example 1 by considering the error of the heuristic function at the start state.

For a heuristic function h, a search problem P, and a corresponding start state s_i, we define the error of h as the ratio between OPT_P and $h(s_i)$. The error of h for a random search problem P_D is itself a random variable, denoted $\frac{\text{OPT}}{h(s_i)}(P_D)$. The statistics needed for using the h-RATIO condition is the CDF of this random variable, denoted by $F_R(v) = Pr(v \leq \frac{\text{OPT}}{h(s_i)}(P_D))$. As in the ABSOLUTE condition, we define a threshold value $T_R(\epsilon, \delta)$

$$T_R(\epsilon, \delta) = \underset{v \geq 0}{\text{argmax}} \left(F_R\left(\frac{v}{1+\epsilon}\right) \geq 1 - \delta \right)$$

The h-RATIO condition is satisfied if for the incumbent solution U it holds that $U \leq h(s_i) \cdot T_R(\epsilon, \delta)$.

Theorem 2. *The h-RATIO condition is a pBS condition.*

The proof of Theorem 2 is a straightforward adaptation of the proof of Theorem 1. It is given in Appendix A for completeness.

Similar to the ABSOLUTE condition, using the h-RATIO condition in PSF incurs almost no overhead. Simply compute $h(s_i) \cdot T_R(\epsilon, \delta)$ once at the beginning of the search, and from then on just check if U is smaller than it, in which case we can halt the search.

The h-RATIO condition is a special case of a more general pBS condition that considers the likelihood of a search problem having an optimal solution of some value given the heuristic value of the initial state. To implement this condition, we require the cumulative distribution function $F_{h=t}(v) = Pr(\text{OPT} \leq v | h(s_i) = t)$. The corresponding threshold parameter is

$$\underset{v \geq 0}{\text{argmax}} \left(F_{h=h(s_i)}(\frac{v}{1+\epsilon}) \geq 1 - \delta \right)$$

and the resulting stopping condition is to halt the search when the incumbent solution is equal to or smaller than this threshold value. Implementing this pBS condition properly, however, requires a larger training set than the h-RATIO as it requires sufficient problems for training for every heuristic value.

4.3. OPEN-BASED condition

The pBS stopping conditions proposed so far are agnostic to the solution generator being used. Indeed, the only knowledge about the underlying search problem these stopping conditions consider is the statistics generated from the training set and the heuristic of the current start state. While this makes them simpler to implement, this also means that if an incumbent solution U does not satisfy these pBS conditions then the search cannot halt until a new incumbent solution is found. Moreover, these pBS conditions do not consider how the solution generator finds solutions. Thus, they ignore any information discovered about the underlying problem that the solution generator collects during its search. The MAX_{fmin} condition remedies this to some extent, as it considers a lower bound obtained by the solution generator and may halt when the lower bound is increased, but this condition ignores δ and increasing this lower bound can be difficult.

In this section, we propose a pBS condition that monitors the search performed by the solution generator, and based on the information it collects about the underlying search problem may decide to halt the search even if a new incumbent solution has not been found. This pBS condition, called the OPEN-BASED condition, is designed for a specific type of solution generators: solution generators that are based on best-first search. Many anytime search algorithms are best-first searches, including ARA*[18], AWA* [12], and APTS [15]. The OPEN-BASED condition monitors the nodes in OPEN when searching for a new incumbent solution, and decides to halt when it is not likely that any node in OPEN is part of a solution that will show the incumbent solution does not have the desired suboptimality.

We explain the OPEN-BASED condition next in details. First, several definitions are introduced. Let $h^*(n)$ denote the cost of the lowest cost path from n to a goal. Thus, for a search problem P with a start state s_i, we have that $h^*(s_i) = \text{OPT}_P$.

Definition 3 *(Reject).* A node n *rejects* a cost U w.r.t. ϵ if $\left(g(n) + h^*(n)\right) \cdot (1+\epsilon) < U$.

Intuitively, a node n *rejects* a cost U if there is a solution to the underlying search problem that passes through node n and this solution has a cost that is small enough to reject the hypothesis that U has the desired suboptimality $(1+\epsilon)$.

Lemma 1. *If an optimal solution has not been found and every node n in* OPEN *does not reject a solution cost U, then U achieves the desired suboptimality.*

Proof. This proof is by contradiction. Assume that U does not achieve the desired suboptimality, i.e., $U > (1+\epsilon) \cdot \text{OPT}_P$. Let $g^*(n)$ denote the optimal path from the initial state s_i to a node n. Since the optimal solution has not been found, there is a node m in OPEN that is part of an optimal solution in which $g(m) = g^*(m)$ [1]. Thus, $\text{OPT}_P = h^*(s_i) = g(m) + h^*(m)$. Since m is in OPEN and we assumed that all nodes in OPEN do not reject the cost U, it follows that $U \leq \left(g(m) + h^*(m)\right) \cdot (1+\epsilon)$. Thus, we have that $U \leq (1+\epsilon) \cdot \text{OPT}_P$, contradicting the assumption that U does not have the desired suboptimality. □

When a node n is in OPEN, the value of $h^*(n)$ is not known. Thus, determining if a node n rejects a cost U is not feasible. Given appropriate statistics, however, it is possible to estimate the probability that a node rejects a cost U. To this end, we collect and consider statistics about the heuristic error of search nodes encountered by the solution generator. These statistics are similar to those used by the h-RATIO, and are represented by a CDF function $F(n, v) = Pr(\frac{h^*(n)}{h(n)} \leq v)$.

There are several ways to approximate this CDF. In our experiments, we did so by grouping together nodes with the same h value, so that we have a random variable $N_{h=v}$ that represents drawing a node with $h(n) = v$ from the distribution of nodes observed during the search and have an h value equal to v. Then, we approximate $F(n, v)$ with $Pr(\frac{h^*(N_{h=h(n)})}{h(N_{h=h(n)})} \leq v)$, denoted by $F_h(h(n), v)$. See Section 5.2 for details on how we collected information to generate these statistics.

Corollary 1. *The probability that a randomly drawn node rejects a cost U is given by*

$$F_h\left(h(n), \frac{1}{h(n)} \cdot \left(\frac{U}{1+\epsilon} - g(n)\right)\right)$$

Proof. By definition, $F_h\left(h(n), \frac{1}{h(n)} \cdot \left(\frac{U}{1+\epsilon} - g(n)\right)\right)$ is equal to

$$Pr\left(\frac{h^*(N_{h=h(n)})}{h(N_{h=h(n)})} \leq \frac{1}{h(n)} \cdot \left(\frac{U}{1+\epsilon} - g(n)\right)\right) \tag{6}$$

$$= Pr\left((1+\epsilon) \cdot \left(g(n) + h(n) \cdot \frac{h^*(N_{h=h(n)})}{h(N_{h=h(n)})}\right) \leq U\right) \tag{7}$$

which is the probability that n *rejects* U (Definition 3). □

For a given node n we denote the value $\frac{1}{h(n)}(\frac{U}{1+\epsilon} - g(n))$ by $P(U, n, \epsilon)$, and when ϵ is clear from the context we omit it and write $P(U, n)$. Thus, Corollary 1 states that $P(U, n)$ is the probability that node n rejects the cost U.

Definition 4 *(Open-based).* The Open-based condition is satisfied if

$$\sum_{n \in \text{Open}} \log(1 - P(U,n)) \geq \log(1-\delta) \tag{8}$$

The correctness of the Open-based condition relies on the assumption that for any two nodes n and n' in Open the event that n rejects U is not negatively correlated with the event that n' rejects U. Under this assumption, the probability that neither n nor n' will reject U is at least the product of the individual probabilities that this will occur, i.e.,

$$Pr\left(\neg(n \text{ rejects } U) \wedge \neg(n' \text{ rejects } U)\right) \geq (1 - P(U,n)) \cdot (1 - P(U,n')) \tag{9}$$

Theorem 3. *The Open-based condition is a pBS condition if the assumption in Equation (9) holds.*

Proof. If the Open-based condition is satisfied then

$$\sum_{n \in \text{Open}} \log(1 - P(U,n)) \geq \log(1-\delta) \tag{10}$$

$$\prod_{n \in \text{Open}} (1 - P(U,n)) \geq 1 - \delta \tag{11}$$

Due to Equation (9), we have

$$Pr(\forall n \in \text{Open}\ \neg(n \text{ rejects } U)) \geq \prod_{n \in \text{Open}} (1 - P(U,n)) \geq 1 - \delta \tag{12}$$

Following Lemma 1, this means U has the desired suboptimality with the required confidence. □

Note that the Open-based condition could have been defined as in Equation (11), i.e., without the logarithm. We describe it with the logarithm to avoid precision issues.

Algorithm 2: Open-based pBS condition update rule.

```
   Input: m, the node that was currently expanded
 1 if m is a goal node and U is decreased then
 2     P̂(U) ← 0 ;
 3     foreach node m' ∈ Open do
 4         P̂(U) ← P̂(U) + log(1 − P(U, m'));
 5     end
 6 else
 7     P̂(U) ← P̂(U) − log(1 − P(U, m)) ;
 8     foreach child m' of m do
 9         if g(m') updated when m is expanded then
10             if m' was previously in Open then
11                 P̂(U) ← P̂(U) − log(1 − P_old(U, m')) ;
12             end
13             P̂(U) ← P̂(U) + log(1 − P(U, m')) ;
14         end
15     end
16 end
```

A significant advantage of the Open-based condition over the Absolute and h-ratio conditions is that it may be satisfied after expanding a node in Open even if Max_{fmin} and U were not changed. Thus, it is especially important to check this condition efficiently. The runtime required to check if the Open-based condition is satisfied is dominated by the runtime required to calculate $\sum_{n \in \text{Open}} \log(1 - P(U,n))$. Let $\hat{P}(U) = \sum_{n \in \text{Open}} \log(1 - P(U,n))$. The straightforward way to compute $\hat{P}(U)$ is to iterate over all the nodes in Open. Computing $\hat{P}(U)$ in this way after every node is expanded is clearly not efficient when Open is large. Fortunately, $\hat{P}(U)$ can be computed incrementally in a more efficient manner: when a node m is removed from Open we decrease $\hat{P}(U)$ by $\log(1 - P(U,m))$ and when a node m is added to Open we increase $\hat{P}(U)$ by $\log(1 - P(U,m))$. The details of this incremental computation of $P(U,n)$ are given in Algorithm 2 and are explained next.

Let m be a non-goal node selected for expansion. When m is expanded it is removed from Open, and therefore $\hat{P}(U)$ must decrease by $\log(1 - P(U,m))$ (line 7 in Algorithm 2). The nodes generated by m may be added to Open, so we need to update $\hat{P}(U)$ accordingly. Specifically, a generated child m' falls into one of the following cases:

Case 1: m' is not in OPEN or in CLOSED. In this case, m' is added to OPEN. Consequently, $\hat{P}(U)$ is increased by $\log(1 - P(U, m'))$ (line 7).

Case 2: m' is already in OPEN. When generating a node that is already in OPEN its g value may get updated if the path to it through m has a lower cost than its current g value. If $g(m')$ does not change, then $\hat{P}(U)$ also remains unchanged. If $g(m')$ did change, then $\hat{P}(U)$ is updated as follows. Let $P_{old}(U, m')$ and $P_{new}(U, m')$ denote the value of $P(U, m')$ as calculated by the old and new g value, respectively. After generating m' we update $\hat{P}(U)$ by removing $P_{old}(U, m')$ (line 7) and adding $P_{new}(U, m')$ (line 13).

Case 3: m' is in CLOSED (i.e., it was previously expanded). In this case, the g value of m' may still be updated when m was expanded, in cases where the cost of the path to m' through m is lower than $g(m')$. The node m' is then reinserted into OPEN and so we need to add $P_{new}(U, m')$ to $\hat{P}(U)$.[2]

Updating $\hat{P}(U)$ in this incremental manner incurs only $O(1)$ overhead every time a node is generated. In contrast, when a goal node is expanded and a better incumbent solution is found, U decreases. Consequently, when calculating $\hat{P}(U)$, the value $\log(1 - P(U, n))$ must be updated for every node n in OPEN (lines 1–5). This requires $O(|\text{OPEN}|)$ overhead. However, this occurs only when a new incumbent solution is found. If the number of times the incumbent solution is updated is D, then the overhead of updating $\hat{P}(U)$ can be amortized over the cost of generating each node in OPEN, incurring an additional D operations per generated node.

5. Experimental results

In preliminary work on pBS search, we evaluated the pBS search framework and several pBS conditions on the well-known 15-puzzle. Here, we extend our evaluation to four additional domains, which we describe next. All the source code used to run our experiments is publicly available. See Appendix D for details on how to obtain and run it.

5.1. Domains

The first two domains we used (pancakes and grid-based pathfinding) are standard heuristic search benchmarks. The other two domains (vacuum cleaner and dockyard robot) are inspired by classical planning problems and have previously been used by Thayer and Ruml [4] and others [5,20] for evaluating search algorithms. For brevity, we call these domains Pancakes, Pathfinding, Vacuum, and Dockyard.

5.1.1. The pancakes puzzle (Pancakes)

In this domain there are k pancakes of different sizes, represented by a unique number between 1 and k. A state in this domain is a pile of these k pancakes, represented as a permutation of the numbers $1, \ldots, k$. There is a single goal state, which is the state in which the pancakes are sorted from largest to smallest. There are $k-1$ state transition operators, where operator i reverses the first i pancakes in the permutation. The heuristic we used in this domain is the GAP heuristic [21], which adds 1 for every two adjacent pancakes that are not consecutive, e.g., the heuristic for the 4-pancake state $(1, 3, 2, 4)$ is 2, due to the "gap"s after pancakes 1 and 2. In our experiments we experimented with the 40-pancake puzzle. Problems were generated randomly by performing 1000 random flips from the goal state.

5.1.2. Grid-based pathfinding (Pathfinding)

In this domain we are given a grid and two cells in it and the task is to find a path from one cell to the other. As a grid, we used the `brc202d` map from the popular "Dragon Age: Origins" video game, which is publicly available from the `movingai.com` repository [22]. A state is a cell in this grid and we allowed transitions in the four cardinal directions. Problems were generated by randomly selecting two cells in the given map. The heuristic we used for this domain is Manhattan distance. Grid pathfinding is significantly different from the pancakes puzzle in that the graph that represents the state space is given explicitly as input, while the graph that represents the state space of the 40-pancakes puzzle is given implicitly by the start state and the allowed set of operators. As a result, the pancake state-space is combinatorially large. Naturally, this means solving grid pathfinding problems is, in general, easier than solving the pancake puzzle.

5.1.3. The vacuum cleaner (Vacuum)

This domain was inspired by the first state-space presented in Russell and Norvig's textbook [23]. A vacuum cleaner is working in a grid (200×200) with obstacles (35% of the cells) and there are a number of dirty spots. The cleaner should find a tour that cleans all dirty spots. This problem is a variant of the traveling salesman problem (TSP) and therefore we used a heuristic based on the *minimum spanning tree*.[3] Problems were generated by randomly putting the vacuum cleaner and 5 dirty locations on a 200×200 grid with obstacles. The number and location of the obstacles in the grid was different in each problem, and were set randomly as well.

[2] There are search algorithms that do not reinsert nodes in Case 3. For example, Anytime Repairing A* [18] stores such nodes in a separated list (called the *inconsistent list*) for future usages. In such cases, one could compute $\hat{P}(U)$ over the union of OPEN and the inconsistent list.

[3] There is a variant in which the robot becomes heavier when carrying dirt, but we used the unit-cost version of this domain.

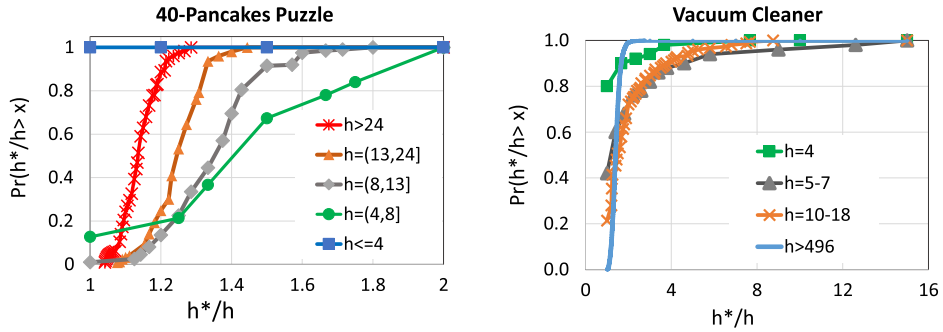

Fig. 2. A sample of the cumulative distribution functions (CDF) of the h^*/h-ratios generated from our training set for the 40-Pancakes puzzle (left) and the Vacuum cleaner domain (right). The different lines in the plots correspond to CDFs generated for different ranges of h values. This plot shows that the heuristic function in both domains is accurate for small h and for very large h values, and less accurate for the h values between these extremes.

5.1.4. The dockyard robot (Dockyard)

This domain was inspired by Ghallab et al. [24] and by the depots domain from the International Planning Competition (IPC). The task is to move containers in the dockyard from one location to another. Similar to the classic blocksworld domain, containers are stacked on different piles and only the topmost container on a pile may be accessed at any time. Stacking and unstacking a container on/from a pile is done by a crane. Moving a container X from pile Y to pile Z involves several actions: unstacking X from Y using a crane, loading X to a mobile robot, moving from the location of Y to the location of Z, unloading X from the robot, and finally stacking X onto Z using a crane. Each action is associated with a cost. Stacking or unstacking from a pile costs 5 plus the height of the pile. Loading or unloading a container from a robot costs 1. Moving a robot between piles costs the distance between the piles. See Thayer and Ruml [4] for the details of how the heuristic is computed. The problems we generated had 5 cranes, 8 containers stacked onto 5 piles, and the average distance between piles is 5.

5.2. Statistics collection

In each of these 4 domains we used 50 problems as the training set for generating the statistics required by the various pBS conditions. The remaining problems (40 problems in the dockyard domain and 50 problems in all other domains) were used to evaluate the performance of the different pBS conditions and search algorithms. This set of problems is referred to as the *testing set*.

The distribution needed by the Absolute condition ($F(v)$) was created by optimally solving all problems in the training set. For the h-ratio condition, we also computed the h value of the start states of these problems. Generating the distribution needed by the Open-based condition ($F_h(h(n), v)$) is more complex as it requires gathering statistics from states observed during the search. To obtain such statistics we run APTS on each of the problems in the training set until it found the optimal solution. Then, we sampled states generated during these APTS runs to obtain states from a range of h values. For each of these states we computed the h^*/h value by finding the optimal path from them to their corresponding goal. The resulting data set of states with their h and h^*/h values was grouped into bins according to their h values such that each bin contains at least 50 problems and the average h^*/h value in each bin is significantly different from the average h^*/h values. Appendix C describes the details of this distribution generation process in more details.

To illustrate the outcome of this process, Fig. 2 shows several cumulative distribution functions generated by this process for Pancakes (left) and Vacuum (right). In both plots the x axis is the h^*/h ratio and the y axis is the cumulative distribution function (CDF) for this ratio, computed as the portion of the problems in the training set for which h^*/h was smaller than the x axis value. As noted above, we grouped together states with close h values. This is shown in Fig. 2, where every curve corresponds to the h^*/h CDFs generated for the different ranges of h values.

The importance of generating different CDFs for different ranges of h values is evident from these plots: in both domains, we observe that when h is small or very large, it tends to be more accurate than when h is between these extremes. For example, in the Pancakes domain when the heuristic is equal to or smaller than 4, it is very accurate. But if h is between 4 and 8, then in approximately 80% of the problems in the training set the heuristic was wrong by more than 20%, i.e., $h^*/h \geq 1.2$. Understanding why the GAP heuristic is more accurate when h is small and when h is large requires a deeper study of the domain and the GAP heuristic.

5.3. Results

In this batch of experiments, we evaluated the basic pBS search framework, which uses an off-the-shelf anytime search algorithm and decides to halt according to one of the proposed pBS conditions: Absolute, h-ratio, and Open-based. The anytime search algorithm we used is APTS, since it is simple to implement and provides state-of-the-art results [15]. We

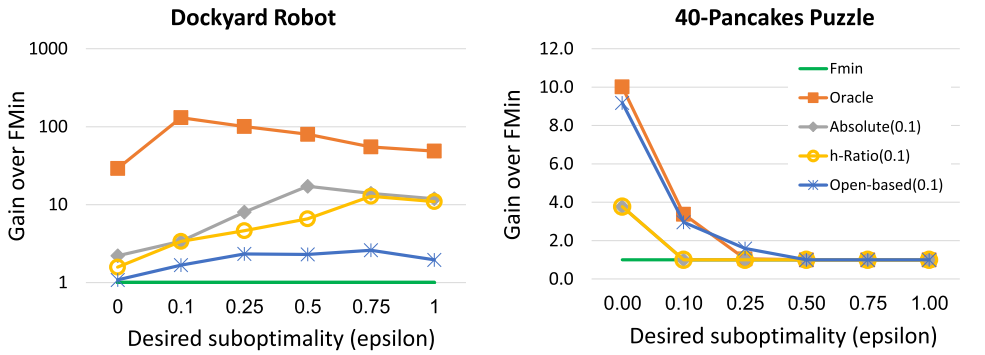

Fig. 3. The gain in terms of number of nodes expanded over using the MAX_{fmin} stopping condition for the Dockyard (left) and the Pancakes (right). The x-axis shows the ϵ values and δ was set to 0.1 in this experiment.

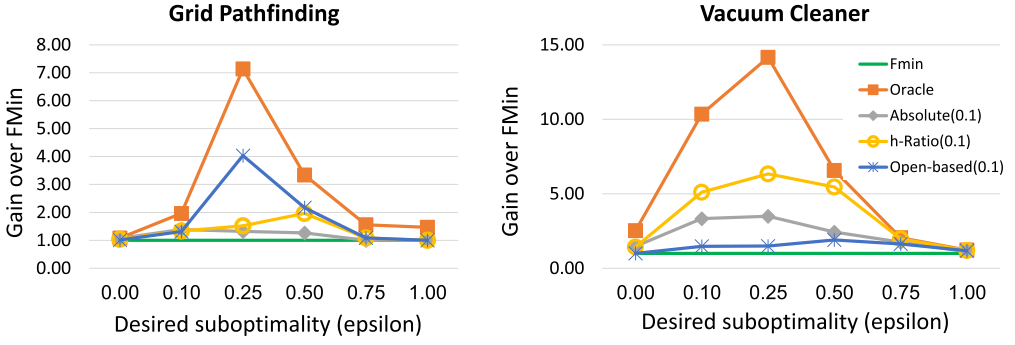

Fig. 4. The gain in terms of number of nodes expanded over using the MAX_{fmin} stopping condition for the Pathfinding (left) and Vacuum (right). The x-axis shows the ϵ values and δ was set to 0.1.

compared the proposed pBS conditions (ABSOLUTE, h-RATIO, and OPEN-BASED) against two baseline stopping conditions: the MAX_{fmin} condition (described in Section 4.2.1) and the ORACLE stopping condition, which is described next.

The ORACLE stopping condition halts the solution generator when the suboptimality of the incumbent solution is equal to or lower than the desired suboptimality. This means ORACLE halts when the incumbent solution is equal to or smaller than $(1+\epsilon)$ times the optimal solution. Thus, to use the ORACLE stopping condition, we first solve optimally the search problem at hand. ORACLE is given only for comparison purposes, as it is not a practical stopping condition, since it requires solving the given problem optimally. Both baseline conditions (ORACLE and MAX_{fmin}) serve, to some extent, as a lower and upper bound of what can be done without considering δ: to find solutions with suboptimality $1+\epsilon$ one can always use the MAX_{fmin} stopping condition, and one cannot hope to guarantee $1+\epsilon$ suboptimality (using the same anytime search) by expanding fewer states than when using the ORACLE stopping condition.

5.3.1. Algorithm runtime comparison

It is common in the heuristic search literature to compare the runtime of search algorithms by comparing the number of nodes they expand. Such a comparison is valid when the runtime of expanding a node is approximately constant and equal for the evaluated algorithms. In our case, the added runtime due to using the h-RATIO and the ABSOLUTE conditions is negligible, since they add a single check after a goal is found. The added runtime of using the OPEN-BASED condition is also not large, as it only adds a non-constant overhead when a new incumbent solution is found, which does not occur frequently in our domains. Indeed, we observed experimentally that average time per node expansion for the evaluated algorithms and stopping conditions was almost the same (the differences were negligible).

Therefore, we compared the runtime performance of the different algorithms by comparing the number of nodes expanded until a solution has been found. Specifically, we computed the ratio between the number of nodes expanded when using the baseline MAX_{fmin} condition and when using each of the evaluated pBS conditions. We refer to this ratio as the *gain* of the evaluated condition. Figs. 3 and 4 show the average gains obtained for the different pBS conditions and domains for $\delta = 0.1$ and $\epsilon = 0, 0.1, 0.25, 0.5, 0.75$, and 1.0.[4] The x-axis is the desired suboptimality ϵ and the y-axis is the aforementioned gain over MAX_{fmin}.

[4] Note that it is possible to set ϵ for values higher than 1, but in our domains the chosen ϵ values were sufficient to display the trends of interest.

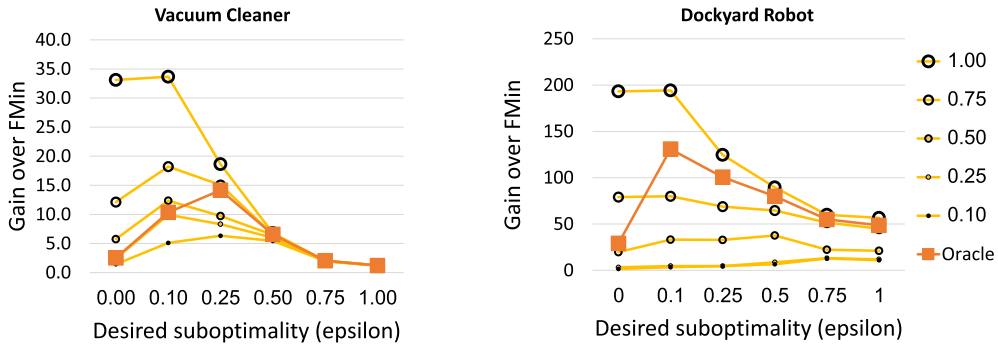

Fig. 5. The impact of changing δ when using the h-ratio pBS condition. Results are shown for Vacuum (left) and Dockyard (right), with the same axes as in Fig. 3. One curve shows the results of the Oracle condition and the other curves show the results of the h-ratio condition using different values of δ.

First, we can see that all three pBS conditions – Absolute, h-ratio, and Open-based – provide some gain over Max_{fmin} in many cases. In each domain there was at least one pBS condition that had a gain of at least 2 for some value of ϵ. For example, in Dockyard both Absolute and h-ratio provide a factor of 10 speedup over just using Max_{fmin}.

In all domains except Dockyard, from a certain value of ϵ the gain over Max_{fmin} diminishes and eventually all pBS conditions converge to the same performance as Max_{fmin}. This is because for values of ϵ that are sufficiently high Max_{fmin} will be sufficient to halt the search immediately after the first solution is found. In particular, if the $h(s) \cdot (1+\epsilon)$ is equal to or larger than the first solution found by the anytime search algorithm then all pBS conditions, including Optimal, will expand exactly the same number of states. Thus, the rate at which the gain over Max_{fmin} decreases to 1 as ϵ increases depends on the accuracy of the heuristic. This explains why the gain of all pBS conditions converged very quickly to 1 in Pancakes (even for ϵ as small as 0.5), since the heuristic we used for this domain (the GAP heuristic) is known to be very accurate and the optimal solutions to all tested problems is no larger than 40. In contrast, the heuristic used in Dockyard is not very accurate and the optimal solutions are often significantly larger (even a factor of three) from the heuristic of the initial state. Thus, we require larger values of ϵ to converge to Max_{fmin} for Dockyard.

It is not clear from the results which of the three proposed pBS conditions achieves the best results. Open-based performs well on the grid pathfinding domain and on the Pancakes puzzle, where in the latter it almost achieves the performance of Optimal. However, on the other two domains it performs poorly. Absolute and h-ratio perform similarly on most domains, with h-ratio being better in the Vacuum cleaner domain and poorer in the dockyard domain. These results show that none of these pBS conditions dominates the others and a deeper understanding of the underlying domain is needed to identify the best pBS condition to use for a given problem. This is left for future work.

Let us now consider the results for the Oracle stopping condition. As expected, Oracle serves as an upper bound on the performance of all other pBS conditions. However, in quite a few cases it is on-par with some of the other pBS conditions, and in Pancakes for $\epsilon = 0.25$ its gain is slightly smaller than the gain of Open-based. This is because Oracle only returns solutions that are $(1+\epsilon)$-suboptimal while our pBS condition can return solutions that are not $(1+\epsilon)$-suboptimal in $1-\delta$ of the cases. This effect is almost not noticeable in the results discussed so far since they were generated for a small value of δ (0.1).

5.3.2. Changing the required confidence $(1-\delta)$

Next, we analyzed the impact of increasing the δ parameter. Setting δ to be higher than 0.1 leads to cases where our pBS conditions achieve better gains than Oracle. Specifically, we experimented with $\delta = 0.25, 0.5, 0.75$, and 1.00.[5] Fig. 5 shows the gain over Max_{fmin} when using the h-ratio pBS condition, for the Vacuum (left) and Dockyard (right) domains. Each curve shows the results using the h-ratio stopping condition with different δ values. For reference, we added a curve for the Oracle results. As expected, increasing δ causes the h-ratio stopping condition to halt earlier, thereby yielding higher gains in runtime. This verifies a key property of an effective pBS stopping condition: that it considers effectively both ϵ and δ parameters. Moreover, for high enough δ values the results of the h-ratio stopping condition even surpasses those of the Oracle stopping condition. However, this only occurs for δ values higher than 0.25. The same trends discussed above were also observed in the other domains and pBS stopping conditions.

5.3.3. Solution quality in practice

Next, we consider the actual cost of the solutions found by our pBS framework with the different pBS conditions. In particular, we analyzed the actual suboptimality of the found solutions and compared it to the desired suboptimality and required confidence, as follows. For every combination of pBS condition, ϵ, and δ, we computed the actual suboptimality of the solution obtained for every problem in our benchmark. We then calculated the ratio of problems for which the

[5] Note that setting $\delta > 1.00$ is not possible, and setting $\delta = 1.00$ corresponds to halting immediately after the first solution is found.

Table 1
Dockyard: ratio of problems for which the actual solution suboptimality was smaller than a given value (columns), for different pBS conditions and values of ϵ, and $1-\delta = 0.9$ (top table) and $1-\delta = 0.75$ (bottom table).

	Absolute						h-Ratio						Open-based					
$1+\epsilon$	1.00	1.10	1.25	1.50	1.75	2.00	1.00	1.10	1.25	1.50	1.75	2.00	1.00	1.10	1.25	1.50	1.75	2.00
Required confidence $1-\delta = 0.9$																		
1.00	**0.94**	1.00	1.00	1.00	1.00	1.00	**0.94**	1.00	1.00	1.00	1.00	1.00	**1.00**	1.00	1.00	1.00	1.00	1.00
1.10	0.60	**1.00**	1.00	1.00	1.00	1.00	0.46	**0.98**	1.00	1.00	1.00	1.00	0.76	**1.00**	1.00	1.00	1.00	1.00
1.25	0.12	0.84	**1.00**	1.00	1.00	1.00	0.02	0.64	**0.98**	1.00	1.00	1.00	0.18	1.00	**1.00**	1.00	1.00	1.00
1.50	0.00	0.40	0.86	**1.00**	1.00	1.00	0.00	0.10	0.78	**1.00**	1.00	1.00	0.00	0.62	1.00	**1.00**	1.00	1.00
1.75	0.00	0.06	0.76	1.00	**1.00**	1.00	0.00	0.04	0.64	1.00	**1.00**	1.00	0.00	0.18	0.88	1.00	**1.00**	1.00
2.00	0.00	0.04	0.66	1.00	1.00	**1.00**	0.00	0.04	0.62	0.98	1.00	**1.00**	0.00	0.06	0.70	1.00	1.00	**1.00**
Required confidence $1-\delta = 0.75$																		
0.00	**0.76**	0.94	1.00	1.00	1.00	1.00	**0.78**	0.98	1.00	1.00	1.00	1.00	**1.00**	1.00	1.00	1.00	1.00	1.00
0.10	0.46	**0.84**	1.00	1.00	1.00	1.00	0.36	**0.88**	1.00	1.00	1.00	1.00	0.76	**1.00**	1.00	1.00	1.00	1.00
0.25	0.12	0.68	**0.92**	1.00	1.00	1.00	0.00	0.50	**0.96**	1.00	1.00	1.00	0.18	1.00	**1.00**	1.00	1.00	1.00
0.50	0.00	0.18	0.86	**1.00**	1.00	1.00	0.00	0.08	0.74	**1.00**	1.00	1.00	0.00	0.62	1.00	**1.00**	1.00	1.00
0.75	0.00	0.06	0.72	1.00	**1.00**	1.00	0.00	0.04	0.64	1.00	**1.00**	1.00	0.00	0.18	0.88	1.00	**1.00**	1.00
1.00	0.00	0.04	0.62	0.98	1.00	**1.00**	0.00	0.04	0.60	0.96	1.00	**1.00**	0.00	0.06	0.70	1.00	1.00	**1.00**

actual suboptimality was larger than a fixed value V set to 1.0, 1.1, 1.25, 1.5, 1.75, and 2.0. These results, for the Dockyard domain, are presented in Table 1. Each column corresponds to a combination of pBS condition and value of V. Each row corresponds to a different desired suboptimality $(1+\epsilon)$. The top and bottom tables corresponds to required confidence $(1-\delta)$ of 0.9 and 0.75, respectively. For example, consider the second entry (Absolute pBS condition, $V = 1.10$) in the third row $(1+\epsilon = 1.25)$ of the top table $(1-\delta = 0.9)$. The value in this entry is 0.84. This means that in 84% of the problems, the actual suboptimality obtained when using the Absolute pBS condition, setting $1-\delta = 0.9$ and $1+\epsilon = 1.25$, resulted in a suboptimality that is equal to or greater than 1.10 $(= V)$. Similarly, the next entry to the right indicates that the actual suboptimality of 100% of the problems solved with this configuration was equal to or greater than 1.25.

We highlighted in bold the table entries for which V is equal to the desired suboptimality $(1+\epsilon)$. By Definition 1, a pBS search algorithm should have a value of at least $1-\delta$ in these entries. As can be seen, this holds for all values of ϵ and δ we experimented with and all the proposed pBS conditions. In fact, the values of the bold entries are usually much larger than their corresponding $1-\delta$ values. We observed similar behavior in all other domains. This suggests that there is room for future research to come up with stronger pBS search algorithm that would halt earlier and still maintain the desired guarantees.

6. ϵ-Aware solution generator

So far, we have considered instances of Psf in which the solution generator is an anytime search algorithm, and completely decoupled the behavior of the search algorithm – which node to expand next – and the stopping condition being used. In this section, we propose a solution generator that is specifically designed to find solutions that satisfy the stopping condition being used.

Consider the Absolute and h-ratio conditions. Both compute a threshold value in the beginning of the search and halt when an incumbent solution is equal to or lower than this threshold. Thus, for Psf instances that use these conditions we propose to use a *bounded-cost* search (BCS) algorithm as a solution generator, setting their cost-bound as the stopping condition's threshold value. In our experiments we used the Potential Search BCS algorithm [15]. However, using a BCS algorithm as a solution generator in such a way is not possible when using the Open-based condition, as it does not have a constant threshold. Developing a solution generator that is specifically tuned for the Open-based condition is a topic for future work.

6.1. Experimental results

We evaluate experimentally all Psf instances, including those that use a BCS algorithm as a solution generator. In addition, we compare the performance of these Psf instances with Dynamic Potential Search (DPS), a recently introduced state-of-the-art bounded-suboptimal search algorithm [5]. Here too, we use the gain over Max_{fmin} (using APTS as a solution generator) as a metric for comparison. In every domain and every value of ϵ, we highlight in bold the best performing algorithm, i.e., the one with the highest gain. Tables 2 and 3 show the gain results for $\delta = 0.1$ for all evaluated algorithms.

The results show that in all domains and across most values of ϵ, the best performing algorithm was a pBS search algorithm. An exception to this is in Pathfinding, where for small values of ϵ DPS outperformed all pBS search algorithms. However, starting from $\epsilon = 0.25$ and higher, a pBS search algorithm has the best performance. As in the results presented earlier, we see that there is no universal winner, and different pBS search algorithms are suitable for different settings. Open-based is particularly useful for Pancakes and Pathfinding, but not so in the other domains. The bounded-cost pBS search performs very well in Dockyard, and in general is quite robust and effective.

Table 2
The gain over MAX_{fmin} for Pancakes and Dockyard, when $\delta = 0.1$. The columns "O", "R", and "A" represent using OPEN-BASED, h-RATIO, and ABSOLUTE, respectively.

Domain search ϵ	Pancakes						Dockyard					
	Anytime			Bounded cost			Anytime			Bounded cost		
	O	R	A	R	A	DPS	O	R	A	R	A	DPS
0.00	**9.18**	3.77	3.77	2.63	2.13	0.84	1.08	1.57	2.21	2.74	**3.13**	1.69
0.10	2.96	1.00	1.00	**5.14**	2.03	2.06	1.67	3.38	3.39	**6.88**	5.34	3.00
0.25	**1.59**	1.00	1.00	1.34	1.36	1.23	2.33	4.62	**8.01**	6.60	6.24	4.17
0.50	**1.00**	**1.00**	**1.00**	0.99	0.98	0.98	2.30	6.58	**17.19**	5.39	6.54	3.64
0.75	1.00	1.00	1.00	**1.01**	**1.01**	1.00	2.60	12.78	13.93	4.54	**16.43**	2.73
1.00	1.00	1.00	1.00	**1.01**	1.00	1.00	1.95	10.94	11.90	14.28	**17.85**	3.18

Table 3
The gain over MAX_{fmin} for Pathfinding and Vacuum, where $\delta = 0.1$. The columns "O", "R", and "A" represent using OPEN-BASED, h-RATIO, and ABSOLUTE, respectively.

Domain search ϵ	Pathfinding						Vacuum					
	Anytime			Bounded cost			Anytime			Bounded cost		
	O	R	A	R	A	DPS	O	R	A	R	A	DPS
0.00	1.01	1.04	1.08	5.65	6.04	**6.43**	1.00	1.43	1.48	**2.75**	2.55	1.38
0.10	1.31	1.34	1.39	3.86	3.95	**4.53**	1.48	5.10	3.33	**6.33**	4.81	3.23
0.25	**4.03**	1.52	1.32	2.45	2.54	2.67	1.49	**6.33**	3.49	5.99	4.52	2.81
0.50	**2.17**	1.96	1.26	1.32	1.26	1.39	1.90	**5.45**	2.43	3.30	1.97	1.69
0.75	**1.09**	**1.09**	**1.00**	0.97	0.93	0.99	1.63	**1.98**	1.75	1.31	1.22	1.10
1.00	**1.00**	**1.00**	**1.00**	0.92	0.90	0.94	1.18	**1.19**	1.11	0.90	0.91	0.88

7. Discussion

The key assumptions in this work are summarized below.

1. Problems observed in the past provide representative statistics for problems that will be given in the future.
2. The search algorithm has access to a representative sample of past problems.
3. It is possible to obtain optimal solutions to these sampled problems.

Here we discuss when these assumptions are reasonable.

7.1. A distribution of problems

As evident from Definition 1, the notion of a pBS search algorithm is only defined with respect to a distribution over a **set** of search of problems. Thus, pBS search is not relevant to cases where there is a single search problem that needs to be solved only once. We argue that many realistic search problems are not such "one-shot" problems. For example, consider navigation applications such as Waze or Google Maps. These applications are constantly solving path-finding search problems, often on the same road map but searching for paths between different source-destination pairs. By analyzing historical data, one can estimate the navigation requests that will need to be handled in the future. This is also true for applications that compute the minimal number of hops between profiles in a social web [25], and for navigating robots in a warehouse [26].

7.2. The availability of a representative training set

Having a representative training set is a common assumption in the analysis of machine learning algorithms. Even in the heuristic search literature, there have been several prior works that assumed having such a representative training set, especially in the context of algorithms that predict search effort [27–30]. Indeed, in some settings having a training set is natural, such as in the pathfinding example mentioned above. In other cases we might be required to generate such a training set by actively sampling the state space. The procedure used to sample the state space needs to be designed so that the distribution of the sampled states is similar to the real distribution of start states. In some domains this may be difficult, while in other domains sampling states from the same distribution is easy. For example, sampling 15-puzzles problems from a uniform distribution over the state space can be done by generating a random permutation of the 15 tiles and verifying mathematically that the resulting permutation represents a solvable 15-puzzle problem [31]. A general approach for sampling states is to perform sequences of random walks from a set of known start states. While random walks are not guaranteed to provide a representative selection of the possible start-goal pairs, using them to generate a sample of the states in the search space is a commonly used approach.

7.3. Solving optimally the training set instances

While some search problems are too hard to solve optimally, many can be solved optimally, but it is very time consuming. Our work focuses on these latter cases, as we can assume then that the time consuming process of solving the training set optimally is done in a preprocessing step. Performing a costly preprocessing step in order to speed up the runtime of solving future problems is common in search algorithms [32] and other fields. Future work will investigate how to handle cases where we cannot even solve the training set problems.

8. Related work

This work on pBS search is related to several previously studied topics. First, we discuss the relation between pBS search and PAC learning [8], and learning in heuristic search (Section 8.1). Then, we discuss the relation between pBS search and Pearl and Kim's notion of δ-risk admissibility [3] (Section 8.2). Finally, we discuss works in other fields that consider a similar form of probably bounded-suboptimal solution guarantee (Section 8.3).

8.1. Learning in heuristic search

The notion of pBS search was inspired by the notion of PAC learning [8], and earlier versions of this work referred to pBS search as PAC search. The terminology change from PAC search to pBS search was done to emphasize that pBS search and PAC learning are significantly different. A learning algorithm is called a *PAC learning algorithm* if with probability higher than $1-\delta$ it learns a classifier that is correct with probability higher than $1-\epsilon$. The $1-\delta$ part of the definition refers to the probability of drawing representative samples **during** training, and the $1-\epsilon$ part refers to the probability that **after** training the classifier we will draw an instance that will be classified correctly. In pBS search, both parameters – ϵ and δ – refer to what happens after "training": $1-\delta$ is the probability of drawing a problem for which a pBS search algorithm will output a solution that is at most $1+\epsilon$ times the optimal solution. Thus, one may say that the δ in PAC learning corresponds to the ϵ in pBS. Also, there is no notion of "solution" quality in PAC learning – either an instance is classified correctly or not. As such, there is no true equivalent in PAC learning to the ϵ parameter in pBS search.

The possible connection between the PAC learning framework and heuristic search have been previously noted in the literature by Ernandes and Gori [33]. They used an artificial neural network to generate a heuristic function $\hat{h}$ that is only *likely admissible*, i.e., admissible with high probability. Experimentally, they showed that A* with $\hat{h}$ as its heuristic can solve the 15-puzzle quickly and return the optimal solutions in many cases. This can be viewed as a special case of the pBS search concept, where $\epsilon = 0$ and $\delta > 0$.

In addition, they bounded the quality of the returned solution to be a function of two parameters: 1) $P(\hat{h} \uparrow)$, which is the probability that $\hat{h}$ is overestimating the optimal cost, and 2) d, the length (number of hops) of the optimal path to a goal. Specifically, the probability that the path found by A* with $\hat{h}$ as a heuristic is optimal is given by $(1-P(\hat{h}\uparrow))^d$. However, this formula is only given as a theoretical observation. In practice, the length of the optimal path to a goal d is not known until the problem is solved optimally, and thus this bound cannot be used to identify whether a solution is probably optimal in practice.

Other search algorithms that use machine learning techniques to generate accurate heuristics have also been proposed [34,35]. Samadi et al. [34] also used an artificial neural network to learn an accurate heuristic, using the values of other heuristics as features. This heuristic was then used by an A* search. To improve the quality of the solutions found by A* with the learned heuristic, the learning process was biased towards generating an admissible heuristic. Experimental results have shown that indeed the quality of the solutions found by A* with the learned heuristic was close to optimal. While the learned heuristic was shown to be highly effective in practice, no theoretical analysis was given on the amount of suboptimality achieved.

For large state spaces where it is hard to obtain training examples, a bootstrapping approach was proposed [35]. An initial heuristic is learned from examples that can be solved easily. Following, the learned heuristic is used to search for solutions for harder examples. The found solutions are then used as a training set for learning an improved heuristic. This process is repeated, leading to learning heuristics of increasing accuracy. The final heuristic generated by this process is the current state-of-the-art heuristic for several domains [35]. For this work as well, the learned heuristic was shown to be very effective in practice, but no theoretical analysis was given on the amount of suboptimality achieved.

8.2. δ-Risk admissibility

The concept of pBS search is reminiscent of the δ-risk admissibility concept defined in the seminal work on semi-admissible heuristic search by Pearl and Kim [3].

They defined a δ-risk admissible search algorithm as an algorithm that, informally, bounds by δ the *risk* that the solution it returns is not optimal. To quantify *risk*, they introduced the concept of a *risk function* and proposed several possible risk functions. A δ-risk admissible search algorithm is defined with respect to a specific risk function as a search algorithm that can halt only if every node in OPEN has a risk that is not larger than δ. They also introduced an actual δ-risk-admissible algorithm called R^*, which is a best-first search that expands nodes according to their risk.

One of the risk functions proposed by Pearl and Kim is the probability that a node n has $f^*(n) < C$, where C is the incumbent solution cost. While appearing similar, even for this risk function there is a major difference between a pBS search algorithm and a δ-risk-admissible algorithm. A δ-risk-admissible algorithm must verify that

$$\forall n \in \text{OPEN} \;\; Pr(f^*(n) < C) < \delta$$

while a pBS search algorithm needs to verify that

$$Pr(\bigvee_{n \in \text{OPEN}} f^*(n) < C) < \delta.$$

As such, a δ-risk-admissible algorithm is not necessarily a pBS search algorithm and the R^* algorithm is not a pBS search.

8.3. Probably suboptimal guarantees in related fields

To the best of our knowledge, we are the first to characterize and propose heuristic search algorithms that guarantee to return a solution that is bounded-suboptimal with high probability. This kind of guarantee has been discussed, however, in other fields. For example, the term *probably approximately correct* search has been used in the information retrieval field for the problem of searching for a document collection in a set of independent computers [36]. A "correct search" was defined as the result of a deterministic search over the set of computers. A randomized distributed search algorithm was proposed that returns search results that are approximately correct with high probability.

Another example where a probably suboptimal guarantee was discussed is in the field of experiment planning, in the context of a problem called the *satisficing search problem* [37]. This problem consists of a set of experiments that can be performed in some order. Each experiment has a cost and a set of possible outcomes (e.g., fail or success). The distribution over the possible outcomes of every experiment is given, and the goal is to obtain a satisfying configuration of experiment outcomes. The task is to build a strategy for choosing which experiments to perform. A *probably approximately optimal* algorithm was proposed, which returns a strategy that is with high probability approximately optimal in terms of the expected experiment cost.

A somewhat related topic from complexity theory is *approximation algorithms* and *probabilistically checkable proofs (PCP)* [38,39]. A key question in complexity theory is which problems can be efficiently approximated, i.e., whether there is an efficient algorithm that is guaranteed to return a solution whose suboptimality is bounded. Complementing this question is whether the correctness of a solution to a problem can be verified probabilistically by having only part of the proof of its correctness. The relation to pBS search is that a bounded-suboptimal search algorithm is, in fact, an approximation algorithm (albeit possibly not efficient), and verifying probabilistically that a solution is optimal corresponds to our pBS guarantee.

9. Conclusion and future work

In this paper we introduced a novel form of bounded-suboptimality for search algorithms called probably bounded-suboptimal search (pBS search). A pBS search algorithm provides control over the suboptimality of solutions it finds as well as the confidence of achieving that suboptimality. A general framework named PSF was introduced as a basis for the development of pBS search algorithms, and several instances of this framework have been proposed. Key in the development of these instances is the use of statistics generated from analyzing a training set of problems and their optimal solutions. Some of the PSF instances we propose only require a representative sample of problems while others require more involved statistics about the nodes generated during the search. A thorough experimental evaluation shows that there is no PSF instance that dominates the others, and the success depends on properties of the domain.

Since this is the first work on this subject, there are many possible directions for future work. Some of these directions were already mentioned in Section 7. Another interesting direction for future work is to consider other state features as meaningful statistics for predicting the heuristic error. For example, in pathfinding, the Manhattan distance heuristic is not accurate in areas with many obstacles but it is extremely accurate in open spaces.

We introduced several pBS conditions and proved that each holds individually, but we have not explored how pBS conditions can be combined. A naive combination of several pBS condition is in the form of a disjunction, i.e., halt when any of the rules says to do so. The resulting pBS condition, however, is not guaranteed to satisfy the same suboptimality with the same confidence, since the different pBS conditions may accept solutions with suboptimality higher than the desired suboptimality in different problems. However, future work can explore how to construct an ensemble of pBS conditions in a valid way.

Also, a deeper inspection of the threshold-based pBS conditions shows that they often exhibit more runtime improvements for "easier" instances rather than "harder" ones. To see this, suppose that we have computed a threshold of $T(\delta, 0)$ for some $\delta > 0$. Recall that this intuitively means that the proportion of problems in the training set with an optimal cost of more than $T(\delta, 0)$ is δ. Now consider a problem with an optimal solution cost larger than $T(\delta, 0)$. Using a pBS search algorithm that uses the ABSOLUTE pBS condition to solve this problem for the same δ and $\epsilon = 0$ will end up halting only when the MAX_{fmin} condition is true. This only happens once the problem is proven to be solved optimally, so there will be

no speedup on this problem compared to the baseline case $\delta = 0$. In contrast, the Absolute pBS condition may halt earlier for problems whose optimal solution is smaller than $T(\delta, 0)$. Thus, the Absolute pBS condition is more effective for problems with smaller optimal solution cost, which are, generally, the problems that are easier to solve. A similar effect occurs for the h-ratio pBS condition for problems in which the heuristic value of the start state is significantly smaller than the optimal solution cost. In other words, these pBS conditions risk missing the desired suboptimality in the easier problems, while always achieving it for the harder problems. While this is desirable in some cases, it is not so in other cases. Future research is needed to devise pBS conditions that control this bias.

An exciting line of future work is to explore search strategies and heuristics that are specifically designed for pBS search. All the Psf instances we proposed used either an admissible heuristic or no heuristic at all, and followed a fairly standard best-first search framework. However, one can develop pBS conditions that consider inadmissible heuristics and statistics about it instead of, or in addition to, an admissible heuristic. This is especially interesting because there are modern methods that use machine learning to generate heuristics that are accurate yet inadmissible [35]. In addition, one can consider search strategies that are not best-first search as solution generators. For example, depth-first branch and bound is a depth-first anytime search algorithm that is known to be very effective in many domains. An open question is how to implement stopping conditions like the Open-based condition or Max_{fmin} with this algorithm.

Acknowledgements

The authors wish to thank Ariel Felner and Robert Holte for their assistance and advice in preliminaries version of this work. We also thank Daniel Gilon, Vitali Sepetnisky, and Mathew Hatem, for providing their source code, which was the basic framework on top of which we implemented and ran our experiments.

Appendix A. Correctness of the h-ratio condition

The proof of Theorem 2 follows the same lines as the proof of Theorem 1, and is given here for completeness.

Proof. Let P be a problem drawn from $\mathcal{P}$ according to distribution D, and let A be an instance of Psf that uses the h-ratio condition. When A halts on P then either the Max_{fmin} condition or the h-ratio condition is satisfied. If the former condition is satisfied then the incumbent solution is guaranteed to have the desired suboptimality, for any value of δ. If A halted due to the h-ratio condition, it means that $cost(A, P) \leq h(s_i) \cdot T_R(\epsilon, \delta)$, and consequently, $\frac{cost(A,P)}{h(s_i)} \leq T_R(\epsilon, \delta)$. Since $T_R(\epsilon, \delta)$ is defined as the largest value v for which $F_R(\frac{v}{1+\epsilon}) \geq 1 - \delta$, it follows that $F_R\left(\frac{cost(A,P)}{h(s_i)\cdot(1+\epsilon)}\right) \geq 1 - \delta$. By definition of F_R we have that

$$Pr\left(\frac{cost(A, P)}{h(s_i) \cdot (1 + \epsilon)} \leq \frac{\text{Opt}}{h(s_i)}(P_D)\right) \geq 1 - \delta \quad \square$$

Appendix B. The incorrectness of the lower-bounded ratio-based pBS condition

In preliminary work on pBS search (referred to there as PAC search) [7], we proposed an additional pBS condition referred to there as the "Lower-Bounded Ratio-Based PAC Condition". This condition is a refinement of the h-ratio condition, in which instead of considering the value of

$$F_R(\frac{U}{1+\epsilon}) = Pr(\frac{U}{1+\epsilon} \leq \frac{\text{Opt}}{h(s_i)}(P_D))$$

the following was used

$$Pr(\frac{\text{Max}_{fmin}}{h(s_i)} \leq \frac{U}{1+\epsilon} \leq \frac{\text{Opt}}{h(s_i)}(P_D))$$

The intuitive explanation for this is that since Max_{fmin} is a lower bound on Opt, it can also be used to lower bound the suboptimality of U. However, this condition turns out to be incorrect. The reason is that it requires knowing the error distribution of $h(s_i)$ given that a specific value of Max_{fmin} has been found. This creates a coupling between the stopping condition and the underlying search algorithm, and thus using the Max_{fmin} lower bound to improve the h-ratio pBS condition is not a valid pBS condition.

Appendix C. Sampling states with different h values

Here we describe how the distribution used by the Open-based condition ($F_h(h(n), v)$) was generated in our experiments. This process consists three steps: sampling, solving, and binning.

Step 1: Sampling. Let $P_1, \ldots P_{50}$ denote the 50 problems in our training set. We run APTS on each of these problems until it found the optimal solution. Let $H(P_i)$ denote the set of unique h values observed in states generated when solving

P_i, and let $Gen(P_i, h = t)$ denote the set of states with h value equal to t that were generated by APTS when solving problem P_i. For every problem P_i and h value $t \in H(P_i)$ we sampled a single state from $Gen(P_i, h = h_t)$. Thus, if APTS generated a state with $h = 3$ in all the 50 problem then we sampled exactly 50 states with $h = 3$: one from $Gen(P_1, h = 3)$, one from $Gen(P_2, h = 3)$, and so on.

Note that to sample uniformly from $Gen(P_i, t)$ we do not need to actually store all the states in $Gen(P_i, t)$. It is sufficient to maintain throughout the search the number of states generated so far with $h = t$, denoted C_t, and one of these states, denoted s_t. When the first state with $h = t$ is generated, s_t is set to be this state and C_t is set to 1. Afterwards, whenever a state s'_t is expanded with $h(s'_t) = t$, we increment C_t by 1 and set s_t to be that state with probability $\frac{1}{C_t}$. At the end of the search, s_t is exactly a state sampled uniformly from $Gen(P_i, t)$.

Step 2: Solving. For each of the sampled states, we compute h^*/h. Computing h for a sampled state is trivial. To compute h^*, we run an optimal solver. That is, for a state s sampled from $Gen(P_i, t)$ we compute h^* by finding an optimal solution to the search problem in which s is the initial state and the goal is the goal of P_i.

Step 3: Binning. The resulting data set of states and their h^*/h values was grouped into bins according to the h values of the states. This was done in a way that resulted in each bin having at least 50 problems. To avoid having too many bins with similar distributions, we grouped together adjacent bins if the difference between the average h^*/h value in these bins was less than 0.01. Then, we compute for each bin a CDF of the h^*/h values in it. The left-hand side of Fig. 2 shows all the CDFs generated for the Pancakes domain.

Appendix D. Reproducing the experimental results

All the source code used to run the experiments described in this paper is publicly available in the Github repository

`https://github.com/galdreiman/j-PAC-heuristic-search.git`.

The specific version of the code used for the experiments in this paper is under the git tag `aij-final`. To build the source code, use the Gradle build tool (`https://gradle.org/`) and run the command `gradle fatJar`. The resulting JAR file can be run with the following command line arguments:

- **CollectOpenBased.** This generates the statistics used for the OPEN-BASED condition.
- **CollectThresholdBased.** This generates the statistics used for the ABSOLUTE and h-RATIO conditions.
- **Run.** This runs the PSF with APTS as a solution generator and the ABSOLUTE, h-RATIO, and MAX_{fmin} conditions on our problem set.
- **RunOracle.** This runs PSF with APTS as a solution generator and the ORACLE stopping condition on our problem set.
- **RunDPS.** This runs DPS on our problem set.
- **RunOpenBased.** PSF with APTS as a solution generator and the OPEN-BASED condition on our problem set.
- **BoundedCostBased.** PSF with PTS as a solution generator and the and ABSOLUTE and h-RATIO conditions, as explained in Section 6.

References

[1] P.E. Hart, N.J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Syst. Sci. Cybern. 4 (2) (1968) 100–107.
[2] I. Pohl, Heuristic search viewed as path finding in a graph, Artif. Intell. 1 (3) (1970) 193–204.
[3] J. Pearl, J.H. Kim, Studies in semi-admissible heuristics, IEEE Trans. Pattern Anal. Mach. Intell. 4 (1982) 392–399.
[4] J.T. Thayer, W. Ruml, Bounded suboptimal search: a direct approach using inadmissible estimates, in: International Joint Conference on Artificial Intelligence, vol. 2011, IJCAI, 2011, pp. 674–679.
[5] D. Gilon, A. Felner, R. Stern, Dynamic potential search – a new bounded suboptimal search, in: Symposium on Combinatorial Search, SoCS, 2016, pp. 119–123.
[6] R. Stern, A. Felner, R. Holte, Probably approximately correct heuristic search, in: Symposium on Combinatorial Search, SoCS, 2011.
[7] R. Stern, A. Felner, R.C. Holte, Search-aware conditions for probably approximately correct heuristic search, in: The Symposium on Combinatorial Search, SoCS, 2012.
[8] L.G. Valiant, A theory of the learnable, Commun. ACM 27 (1984) 1134–1142.
[9] R.A. Valenzano, S. Jabbari Arfaee, J.T. Thayer, R. Stern, N.R. Sturtevant, Using alternative suboptimality bounds in heuristic search, in: International Conference on Automated Planning and Scheduling, ICAPS, 2013, pp. 233–241.
[10] R.E. Korf, Depth-first iterative-deepening: an optimal admissible tree search, Artif. Intell. 27 (1) (1985) 97–109.
[11] R.E. Korf, Linear-space best-first search, Artif. Intell. 62 (1) (1993) 41–78.
[12] E.A. Hansen, R. Zhou, Anytime heuristic search, J. Artif. Intell. Res. 28 (2007) 267–297.
[13] R. Zhou, E.A. Hansen, Beam-stack search: integrating backtracking with beam search, in: International Conference on Automated Planning and Scheduling, ICAPS, 2005, pp. 90–98.
[14] S. Aine, P. Chakrabarti, R. Kumar, AWA* – a window constrained anytime heuristic search algorithm, in: International Joint Conference on Artificial Intelligence, IJCAI, 2007, pp. 2250–2255.
[15] R. Stern, A. Felner, J. van den Berg, R. Puzis, R. Shah, K. Goldberg, Potential-based bounded-cost search and anytime non-parametric A*, Artif. Intell. 214 (2014) 1–25.
[16] J.T. Thayer, R. Stern, A. Felner, W. Ruml, Faster bounded-cost search using inadmissible estimates, in: International Conference on Automated Planning and Scheduling, ICAPS, 2012, pp. 270–278.

[17] R. Stern, S. Kiesel, R. Puzis, A. Felner, W. Ruml, Max is more than min: solving maximization problems with heuristic search, in: Annual Symposium on Combinatorial Search, SoCS, 2014, pp. 146–156.
[18] M. Likhachev, G.J. Gordon, S. Thrun, ARA*: anytime A* with provable bounds on sub-optimality, in: The Conference on Neural Information Processing Systems, NIPS, 2003, pp. 767–774.
[19] A. Felner, U. Zahavi, R. Holte, J. Schaeffer, N.R. Sturtevant, Z. Zhang, Inconsistent heuristics in theory and practice, Artif. Intell. 175 (9–10) (2011) 1570–1603.
[20] V. Sepetnitsky, A. Felner, R. Stern, Repair policies for not reopening nodes in different search settings, in: The Symposium on Combinatorial Search, SoCS, 2016, pp. 81–88.
[21] M. Helmert, Landmark heuristics for the pancake problem, in: The Symposium on Combinatorial Search, SoCS, 2010, pp. 109–110.
[22] N. Sturtevant, Benchmarks for grid-based pathfinding, IEEE Trans. Comput. Intell. AI Games 4 (2) (2012) 144–148, http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf.
[23] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 3rd edition, Pearson, 2009.
[24] M. Ghallab, D. Nau, P. Traverso, Automated Planning: Theory and Practice, Elsevier, 2004.
[25] R. Bakhshandeh, M. Samadi, Z. Azimifar, J. Schaeffer, Degrees of separation in social networks, in: Symposium on Combinatorial Search, SoCS, 2011, pp. 18–23.
[26] P.R. Wurman, R. D'Andrea, M. Mountz, Coordinating hundreds of cooperative, autonomous vehicles in warehouses, AI Mag. 29 (1) (2008) 9–20.
[27] R.E. Korf, M. Reid, S. Edelkamp, Time complexity of iterative-deepening-A*, Artif. Intell. 129 (1–2) (2001) 199–218.
[28] U. Zahavi, A. Felner, N. Burch, R.C. Holte, Predicting the performance of IDA* using conditional distributions, J. Artif. Intell. Res. 37 (2010) 41–83.
[29] L.H. Lelis, S. Zilles, R.C. Holte, Predicting the size of IDA*'s search tree, Artif. Intell. 196 (2013) 53–76.
[30] L. Lelis, S. Zilles, R.C. Holte, Improved prediction of IDA*'s performance via epsilon-truncation, in: The Symposium on Combinatorial Search, SoCS, 2011, pp. 108–116.
[31] W.W. Johnson, W.E. Story, et al., Notes on the "15" puzzle, Am. J. Math. 2 (4) (1879) 397–404.
[32] R.E. Korf, A. Felner, Disjoint pattern database heuristics, Artif. Intell. 134 (1–2) (2002) 9–22.
[33] M. Ernandes, M. Gori, Likely-admissible and sub-symbolic heuristics, in: European Conference on Artificial Intelligence, ECAI, 2004, pp. 613–617.
[34] M. Samadi, A. Felner, J. Schaeffer, Learning from multiple heuristics, in: AAAI Conference on Artificial Intelligence, 2008, pp. 357–362.
[35] S. Jabbari Arfaee, S. Zilles, R.C. Holte, Learning heuristic functions for large state spaces, Artif. Intell. 175 (16–17) (2011) 2075–2098.
[36] I. Cox, R. Fu, L. Hansen, Probably approximately correct search, in: Advances in Information Retrieval Theory, in: Lecture Notes in Computer Science, vol. 5766, Springer, 2009, pp. 2–16.
[37] R. Greiner, P. Orponen, Probably approximately optimal satisficing strategies, Artif. Intell. 82 (1990) 21–44.
[38] I. Dinur, Probabilistically checkable proofs and codes, in: International Congress of Mathematicians, 2010, p. 265.
[39] S. Arora, S. Safra, Probabilistic checking of proofs: a new characterization of NP, J. ACM 45 (1) (1998) 70–122.