UC Berkeley
UC Berkeley Previously Published Works

Title
Potential-based bounded-cost search and Anytime Non-Parametric A*

Permalink
https://escholarship.org/uc/item/4ct236k1

Authors
Stern, Roni
Felner, Ariel
van den Berg, Jur
et al.

Publication Date
2014-09-01

DOI
10.1016/j.artint.2014.05.002

Peer reviewed

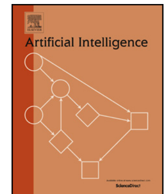
eScholarship.org Powered by the California Digital Library
University of California

Artificial Intelligence 214 (2014) 1–25

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint



Potential-based bounded-cost search and Anytime Non-Parametric A*

Roni Stern [a,*], Ariel Felner [a], Jur van den Berg [b], Rami Puzis [a], Rajat Shah [c], Ken Goldberg [c]

[a] *Information Systems Engineering, Ben Gurion University, Be'er Sheva, Israel*
[b] *School of Computing, University of Utah, Salt Lake City, UT, USA*
[c] *University of California, Berkeley, CA, USA*

ARTICLE INFO

Article history:
Received 1 May 2012
Received in revised form 1 May 2014
Accepted 3 May 2014
Available online 10 May 2014

Keywords:
Heuristic search
Anytime algorithms
Robotics

ABSTRACT

This paper presents two new search algorithms: Potential Search (PTS) and Anytime Potential Search/Anytime Non-Parametric A* (APTS/ANA*). Both algorithms are based on a new evaluation function that is easy to implement and does not require user-tuned parameters. PTS is designed to solve bounded-cost search problems, which are problems where the task is to find as fast as possible a solution under a given cost bound. APTS/ANA* is a non-parametric anytime search algorithm discovered independently by two research groups via two very different derivations. In this paper, co-authored by researchers from both groups, we present these derivations: as a sequence of calls to PTS and as a non-parametric greedy variant of Anytime Repairing A*.

We describe experiments that evaluate the new algorithms in the 15-puzzle, KPP-COM, robot motion planning, gridworld navigation, and multiple sequence alignment search domains. Our results suggest that when compared with previous anytime algorithms, APTS/ANA*: (1) does not require user-set parameters, (2) finds an initial solution faster, (3) spends less time between solution improvements, (4) decreases the suboptimality bound of the current-best solution more gradually, and (5) converges faster to an optimal solution when reachable.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

Heuristic search algorithms are widely used to compute minimum-cost paths in graphs. Applications of heuristic search range from map navigation software and robot path planning to automated planning and puzzle solving. Different search algorithms return solutions of varying quality, which is commonly measured by a cost, where high quality solutions are those with a low cost. Ideally, one would like to find optimal solutions, i.e., those with minimum cost. Given an admissible heuristic, some search algorithms, such as A* [1] or IDA* [2], return optimal solutions. However, many problems are very hard to solve optimally [3] even with such algorithms.

In this paper we propose two algorithms: Potential Search (PTS) and Anytime Potential Search/Anytime Non-Parametric A* (APTS/ANA*). These algorithms are especially suited for cases where an optimal solution is hard to find. PTS is designed to solve problems where any solution with cost less than C, an input to the problem, is acceptable, while solutions with cost $\geq C$ are useless. We call such problems *bounded-cost search problems*. Bounded-cost search problems

* Corresponding author.

http://dx.doi.org/10.1016/j.artint.2014.05.002
0004-3702/© 2014 Elsevier B.V. All rights reserved.

arise, for example, when the expense budget for a business trip is limited and the trip (e.g., flights and hotels) should be planned as quickly as possible but within the budget limit. If an online travel agency such as Expedia or Priceline is planning the trip, computational resources could be diverted to other clients once a plan is found within the budget limits (see Priceline's "Name Your Own Price" option).

The second algorithm we present, APTS/ANA*, is an anytime search algorithm, i.e., an algorithm "whose quality of results improves gradually as computation time increases" [4]. APTS/ANA* can be viewed as a translation of an anytime search algorithm into a sequence of bounded-cost search problems solved by PTS, or as an intelligent approach to avoid the parameter setting problem of Weighted A*-based anytime search algorithms. Setting parameters to bounded-suboptimal search algorithms is a known problem in the heuristic search literature [5]. A key benefit of APTS/ANA* is that it does not require users to set parameters, such as the w_0 and Δw parameters of ARA* [6]. Furthermore, experiments suggest that APTS/ANA* improves upon previous anytime search algorithms in most cases by (1) finding an initial solution faster, (2) spending less time between solution improvements, (3) decreasing the suboptimality bound of the current-best solution more gradually, and (4) converging faster to an optimal solution when reachable.

Both PTS and APTS/ANA* are based on a new evaluation function $u(n) = \frac{C-g(n)}{h(n)}$ that was discovered independently by two research groups via two very different derivations. In this paper, co-authored by researchers from both groups, we present both derivations. The first derivation of $u(\cdot)$ is based on a novel concept called the *potential* of a node. The potential of a node n is defined with respect to a given value C, and is the probability that a node n is part of a solution of cost lower than C. We prove that the node with the highest $u(\cdot)$ is the node with the highest potential, under certain probabilistic relation between the heuristic function and the cost it estimates.

The second derivation of $u(\cdot)$ is based on the desire for a non-parametric version of the Anytime Repairing A* algorithm [6]. We show that expanding the node with the highest $u(\cdot)$ has the same effect as setting the parameters of ARA* dynamically to improve the best solution found so far as fast as possible. In addition, we show that $u(\cdot)$ bounds the suboptimality of the current solution.

We compare PTS and APTS/ANA* with previous anytime algorithms on five representative search domains: the 15-puzzle, robot motion planning, gridworld navigation, Key player problem in communication (KPP-COM), and multiple sequence alignment. As mentioned above, the experimental results suggest that APTS/ANA* improves upon previous anytime search algorithms in terms of key metrics that determine the quality of an anytime algorithm. As a bounded-cost algorithm, our results suggest that PTS, which is specifically designed for bounded-cost problems, outperforms competing algorithms for most cost bounds and exhibits an overall robust behavior.

This paper extends our preliminary work [7–9] by (1) providing a substantially more rigorous theoretical analysis of the presented algorithms, (2) extending the experimental results, (3) adding a comprehensive discussion on the relation between bounded-cost search and anytime search, and (4) discussing the limitations of PTS and APTS/ANA*.

The structure of this paper is as follows. First, we provide background and related work. In Section 3, we introduce the bounded-cost search problem and present Potential Search (PTS). In Section 4, we present Anytime Potential Search (APTS) [7,8] and Anytime Non-Parametric A* (ANA*) [9], showing that they are equivalent and discussing their theoretical properties. Section 5 presents experimental results comparing PTS and APTS/ANA* with previous algorithms. Finally, we discuss a generalization of PTS (Section 6) and conclude with suggestions for future work (Section 7).

2. Background and related work

Search algorithms find a solution by starting at the initial state and traversing the *problem space graph* until a goal state is found. Various search algorithms differ by the order in which they decide to traverse the problem space graph. Traversing the problem space graph involves *generating* and *expanding* its nodes. The term *generating* a node refers to creating a data structure that represents it, while *expanding* a node means generating all its children.

One of the most widely used search frameworks is best–first search (BFS) [10]. BFS keeps two lists of nodes: an *open list* (denoted hereafter as *OPEN*), which contains all the generated nodes that have not been expanded yet, and a *closed list* (denoted hereafter as *CLOSED*), which contains all the nodes that have been previously expanded. Every generated node in *OPEN* is assigned a value by an evaluation function. The value assigned to a node is called the *cost* of the node. In every iteration of BFS, the node in *OPEN* with the lowest cost is chosen to be expanded. This lowest-cost node is moved from *OPEN* to *CLOSED*, and the children of this node are inserted to *OPEN*. The purpose of *CLOSED* is to avoid inserting nodes that have already been expanded into *OPEN*. *CLOSED* is also used to help reconstruct the solution after a goal is found. Once a goal node is chosen for expansion, i.e., it is the lowest-cost node in *OPEN*, BFS halts and that goal is returned.[1] Alternatively, BFS can be defined such that every node is assigned a value, and in every iteration the node with the highest value is expanded.

BFS is a general framework, and many well-known algorithms are special cases of it. For example, Dijkstra's single-source shortest-path algorithm [12] and the A* algorithm [1] are both special cases of BFS, differing only in their evaluation function.[2] Dijkstra's algorithm is BFS with an evaluation function that is $g(n)$, which is the shortest path found so far from

[1] This is the textbook version of BFS [10]. However, there are variants of BFS where the search is halted earlier (e.g., BFS with lookaheads [11]).

[2] In this paper we consider Dijkstra's algorithm in its best–first search variant, which is also known as Uniform Cost Search. It has been shown [13] that this variant of Dijkstra is more efficient than the implementation of Dijsktra detailed in the common algorithm textbook [14].

the start of the search to node n. A* is BFS with an evaluation function that is $f(n) = g(n) + h(n)$, where $h(n)$ is a heuristic function estimating the cost from state n to a goal node. We use the notation $h^*(n)$ to denote the lowest-cost path from node n to a goal. $h(n)$ is said to be *admissible* if it never overestimates the cost of the lowest-cost path from n to a goal, i.e., if $h(n) \leq h^*(n)$ for every node n. Using an admissible $h(n)$, A* is guaranteed to have found a lowest-cost path to a goal when a goal node is chosen for expansion. In general, we use the term *optimal search algorithms* to denote search algorithms that guarantee returning an optimal solution. Other known *optimal search algorithms* include IDA* [2] and RBFS [15].

Finding an optimal solution to search problems is often infeasible. Additionally, it is often the case that non-optimal solutions are good enough. Hence, there are search algorithms that provide a weaker guarantee on the solution they return, with respect to optimal search algorithms. We list below several types of search algorithms that provide such weaker guarantees.

2.1. Bounded-suboptimal search algorithms

Bounded-suboptimal algorithms guarantee that the solution returned is no more than w times the cost of an optimal solution, where $w > 1$ is a predefined parameter. These algorithms are also called w-admissible, and the value of w is referred to as the "desired suboptimality". We use the term *suboptimality* of a solution of cost C to denote the ratio between the cost of an optimal solution and C. Thus, the suboptimality of the solution returned by a w-admissible algorithm is bounded (from above) by the desired suboptimality w.

The most well-known bounded-suboptimal search algorithm is probably Weighted A* (WA*) [16].[3] WA* extends A* by trading off running time and solution quality. It is similar to A*, except that it inflates the heuristic by a value $w \geq 1$. Thus, WA* expands the node n in *OPEN* with minimal $f_w(n) = g(n) + w \cdot h(n)$. The higher w, the greedier the search, and the sooner a solution is typically found. If $h(\cdot)$ is an admissible heuristic, then WA* is a bounded-suboptimal search algorithm, i.e., the *suboptimality* of the solutions found by WA* is bounded by w. Other examples of a bounded-suboptimal search algorithm include A^*_ϵ [18], Optimistic Search [19] and Explicit Estimation Search [20].

2.2. Any solution algorithms

In some cases, the solution quality is of no importance. This may be the case in problems that are so hard that obtaining any meaningful bound on the quality of the solution is not possible. We call algorithms for such settings *any solution algorithms*. Such algorithms usually find a solution faster than algorithms of the first two classes, but possibly with lower quality. Examples of any solution algorithms include pure heuristic search (a BFS using $h(\cdot)$ as its evaluation function),[4] beam search variants [21], as well as many variants of *local search algorithms* such as Hill climbing and Simulated annealing [10].

2.3. Anytime search algorithms

Another class of search algorithms that lies in the range between *any solution* algorithms and *optimal* algorithms is *anytime algorithms*. An anytime search algorithm starts, conceptually, as an any solution algorithm.[5] After the first solution is found, it continues to run, finding solutions of better quality (with or without guarantee on their suboptimality). Anytime algorithms are commonly used in many domains, since they provide a natural continuum between any solution algorithms and optimal solution algorithms. Some anytime algorithms, such as AWA* [22] and ARA* [6] that are introduced below, are guaranteed to converge to finding an optimal solution if enough time is given.

Many existing anytime search algorithms are loosely based on WA*. Since this paper addresses anytime search algorithms in depth, we next provide a brief survey of existing WA*-based anytime search algorithms. A commonly used anytime algorithm that is not related to WA* is Depth-first branch and bound (DFBnB) [23]. DFBnB runs a depth-first search, pruning nodes that have a cost higher than the *incumbent solution* (= best solution found so far). DFBnB does not require any parameters. However, DFBnB is highly ineffective in domains with many cycles and large search depth.[6]

2.4. WA-based anytime search algorithms*

Anytime Weighted A* (AWA*) [22] is an anytime version of WA*. It runs WA* with a given value of w until it finds the first solution. Then, it continues the search with the same w. Throughout, AWA* expands a node in *OPEN* with minimal $f_w(n) = g(n) + w \cdot h(n)$, where w is a parameter of the algorithm. Each time a goal node is extracted from *OPEN*, an improved solution may have been found. Let G denote the cost of the incumbent solution. If $h(\cdot)$ is admissible, then the suboptimality of the incumbent solution can be bounded by $G/\min_{n \in OPEN}\{g(n) + h(n)\}$, as G is an upper bound of the cost of an optimal solution, and $\min_{n \in OPEN}\{g(n) + h(n)\}$ is a lower bound of the cost of an optimal solution. Given enough

[3] The proof for the w-admissibility of WA* is given in the Appendix of a later paper [17]. That paper proposed a variation on WA* (dynamic weighting) but the same proof holds for plain WA*.

[4] Pure heuristic search is sometimes called Greedy best–first search in the literature.

[5] One can use virtually any search algorithm to find an initial solution in an anytime algorithm.

[6] Large solution depth can be partially remedied by applying an iterative deepening framework on top of DFBnB.

runtime, AWA* will eventually expand all the nodes with $g(n) + h(n)$ larger than the incumbent solution, and return an optimal solution.

Anytime Repairing A* (ARA*) [6] is also based on WA*. First, it finds a solution for a given initial value of w. It then continues the search with progressively smaller values of w to improve the solution and reduce its suboptimality bound. The value of w is decreased by a fixed amount each time an improved solution is found or the current-best solution is proven to be w-suboptimal. Every time a new value is determined for w, the $f_w(n)$-value of each node $n \in OPEN$ is updated to account for the new value of w and *OPEN* is re-sorted accordingly. The initial value of w and the amount by which it is decreased in each iteration (denoted as Δw) are parameters of the algorithm.

Restarting Weighted A* (RWA*) [24] is similar to ARA*, but each time w is decreased it restarts the search from the root node. That is, every new search is started with only the initial node in *OPEN*. It takes advantage of the effort of previous searches by putting the nodes explored in previous iterations on a *SEEN* list. Each time the search generates a seen node, it puts that node in *OPEN* with the best g-value known for it. Restarting has proven to be effective (in comparison to continuing the search without restarting) in situations where the quality of the heuristic varies substantially across the search space. As with ARA*, the initial values of w and Δw are parameters of the algorithm.

Anytime Window A* (AWinA*) [25], *Beam-Stack Search* (BSS) [26], and BULB [21] are not based on WA*, but rather limit the "breadth" of a regular A* search. Iteratively increasing this breadth provides the anytime characteristic of these algorithms. For a comprehensive survey and empirical comparison of anytime search algorithms, see [27].

Most existing WA*-based anytime search algorithms require users to set parameters, for example the w factor used to inflate the heuristic. ARA* [6], for instance, has two parameters: the initial value of w and the amount by which w is decreased in each iteration. Setting these parameters requires trial-and-error and domain expertise [25]. One of the contributions of this paper is a non-parametric anytime search algorithm that is based on solving a sequence of bounded-cost search problems (Section 4).

3. Bounded-cost search and potential search

In this section we define the *bounded-cost* search problem, where the task is to find, as quickly as possible, a solution with a cost that is lower than a given cost bound. We then explain why existing search algorithms, such as optimal search algorithms and bounded-suboptimal search algorithms, are not suited to solve bounded-cost search problems. Finally, we introduce a new algorithm called Potential Search (PTS), specifically designed for solving bounded-cost search problems. We define a bounded-cost search problem as follows.

Definition 1 *(Bounded-cost search problem).* Given an initial state s, a goal test function, and a constant C, a bounded-cost search problem is the problem of finding a path from s to a goal state with cost less than C.

3.1. Applications of bounded-cost search

In Section 4 we show that solving bounded-cost search problems can be a part of an efficient and non-parametric anytime search. While constructing efficient anytime algorithms is a noteworthy achievement, solving bounded-cost search problems is important in its own right and has many practical applications.

Generally, when a search algorithm is part of a larger system, it is reasonable to define for the search algorithm what an acceptable solution is in terms of cost. Once such a solution is found, system resources can be diverted to other tasks instead of optimizing other search-related criteria.

For example, consider an application server for an online travel agency such as Expedia, where a customer requests a flight to a specific destination within a given budget. In fact, Priceline allows precisely this option when booking a hotel (the "Name Your Own Price" option). This can be naturally modeled as a bounded-cost problem, where the cost is the price of the flight/hotel.

Furthermore, recent work has proposed to solve conformant probabilistic planning (CPP) problems by compiling them into a bounded-cost problem [28]. In CPP, the identity of the start state is uncertain. A set of possible start states is given, and the goal is to find a plan such that the goal will be reached with probability higher than $1 - \epsilon$, where ϵ is a parameter of the search. We refer the reader to Taig and Brafman's paper [28] for details about how they compiled a CPP problem to a bounded-cost search problem.

A more elaborate application of bounded-cost search exists in the task of recognizing textual entailment (RTE). RTE is the problem of checking whether one text, referred to as the "text", logically entails another, referred to as the "hypothesis". For example, the text "Apple acquired Anobit" entails the hypothesis that "Anobit was bought". Recent work modeled RTE as a search problem, where a sequence of text transformation operators, referred to as a "proof", is used to transform the text into the hypothesis [29,30]. Each proof is associated with a cost representing the confidence that the proof preserves the semantic meaning of the text. Stern et al. [29] used Machine Learning to learn a cost threshold for RTE proofs such that only proofs with cost lower than this threshold are valid. Thus, solving an RTE problem (i.e., determining whether a given text entails a given hypothesis) becomes a bounded-cost problem of finding a proof under the learned cost threshold.

A bounded-cost search problem can be viewed as a constraint satisfaction problem (CSP), where the desired cost bound is simply a constraint on the solution cost. However, modeling a search problem as a CSP is non-trivial if one does not know

the depth of the goal in the search tree.[7] Furthermore, many search problems have powerful domain-specific heuristics, and it is not clear if general CSP solvers can use such heuristics. The bounded-cost search algorithm presented in Section 3.3 can use any given heuristic.

Finally, most state-of-the-art CSP solvers use variants of depth-first search. Such algorithms are known to be highly inefficient in problems like pathfinding, where there are usually many cycles in the state space. Nonetheless, the potential-based approach described in Section 3.3 is somewhat reminiscent of CSP solvers that are based on solution counting and solution density, where assignments estimated to allow the maximal number of solutions are preferred [31].

Another topic related to the bounded-cost setting is resource-constrained planning [32,33], where resources are limited and may be consumed by actions. The task is to find a lowest cost feasible plan, where a *feasible* plan is one where resources are available for all the actions in it. One may view bounded-cost search as a resource-constrained problem with a single resource – the plan cost. However, in bounded-cost search we do not want to minimize the cost of the plan. Any plan under the cost bound is acceptable. Once such a plan is found, it is wasteful to invest CPU time in improving the cost, and computation resources can be diverted elsewhere.

3.2. Naïve approaches to bounded-cost search

It is possible to solve a bounded-cost search problem by running an optimal search algorithm. If the optimal solution cost is less than C, return it; otherwise return *failure*, as no solution of cost less than C exists. One could even use C for pruning purposes, and prune any node n with $f(n) \geq C$. However, this technique for solving the bounded-cost search problem might be very inefficient as finding a solution with cost less than C can be much easier than finding an optimal solution.

One might be tempted to run any of the bounded-suboptimal search algorithms. However, it is not clear how to tune any of the suboptimal algorithms (for example, which weight to use in WA* and its variants), as the cost of an optimal solution is not known and therefore the ratio between the cost of the desired solution C and the optimal cost is also unknown.

A possible direction for solving a bounded-cost search problem is to run any suboptimal best–first search algorithm, e.g., pure heuristic search, and prune node with $g + h \geq C$. Once a solution is found, it is guaranteed to have a cost lower than C. In fact, using this approach with pure heuristic search was shown to be effective in domain independent planning problems with non-unit edge costs [34]. The main problem with all these ad-hoc bounded-cost algorithms is, however, that the desired goal cost bound is not used to guide the search, i.e., C is not considered when choosing which node to expand next.

Having defined the notion of bounded-cost search and its relation to existing search settings and algorithms, we next introduce the PTS algorithm, which is specifically designed to solve such problems.

3.3. The Potential Search algorithm (PTS)

The Potential Search algorithm (PTS) is specifically designed to focus on solutions with cost less than C, and the first solution it finds meets this requirement. PTS is basically a best–first search algorithm that expands the node from *OPEN* with the *largest* $u(\cdot)$, where $u(\cdot)$ is defined as follows[8]:

$$u(n) = \frac{C - g(n)}{h(n)}$$

Choosing the node with the highest $u(\cdot)$ can be intuitively understood as selecting the node that is most likely to lead to a solution of cost less than C, as $u(n)$ is a ratio of the "budget" that is left to find a solution under C (this is $C - g(n)$) and the estimate of the cost between n and the goal (this is $h(n)$). In Sections 3.4 and 4.3 we discuss two analytical derivations of the $u(\cdot)$ evaluation function.

The complete pseudo-code of PTS is provided in Algorithm 1. The input to PTS, and in general to any bounded-cost search algorithm, is the initial state s from which the search begins, and the cost bound C. In every iteration, the node with the highest $u(\cdot)$, denoted as n, is expanded from OPEN (line 3). For every generated node n', duplicate detection is performed, ignoring n' if it exists in *OPEN* or *CLOSED* with smaller or equal g value (line 7). Otherwise, the value of $g(n')$ is updated (line 9). If the heuristic function $h(\cdot)$ is admissible, then PTS prunes any node n' for which $g(n') + h(n') \geq C$ (line 12). This is because these nodes can never contribute to a solution of cost smaller than C. If $h(\cdot)$ is not admissible, then n' is pruned only if $g(n') \geq C$. Any bounded-cost search should perform this pruning, and indeed in our experiments we implemented this for all of the competing algorithms. If n' is not pruned and it is a goal, the search halts. Otherwise, n' is inserted into *OPEN* and the search continues.

3.4. The potential

PTS is based on the novel $u(\cdot)$ evaluation function. Next, we provide the first analytical derivation of $u(n)$, relating it to the probability that n leads to a goal of cost smaller than C. We call this probability the *potential* of a node.

[7] Note that even if the cost bound is C, the search tree might be much deeper if there are edges of cost smaller than one (e.g., zero cost edges).

[8] For cases where $h(n) = 0$, we define $u(n)$ to be ∞, causing such nodes to be expanded first.

Algorithm 1: Potential search.

```
    Input: C, the cost bound
    Input: s, the initial state
 1  OPEN ← s
 2  while OPEN is not empty do
 3  |   n ← argmax_{n∈OPEN} u(n)  // pop the node with the highest u(n)
 4  |   foreach successor n' of n do
 5  |   |   // Duplicate detection and updating g(n')
 6  |   |   if n' is in OPEN or CLOSED and g(n') ≤ g(n) + c(n, n') then
 7  |   |   |   Continue to the next successor of n (i.e., go to line 4)
 8  |   |   end
 9  |   |   g(n') ← g(n) + c(n, n')
10  |   |   // Prune nodes over the bound
11  |   |   if g(n') + h(n') ≥ C (if h is inadmissible, check instead if g(n') ≥ C) then
12  |   |   |   Continue to the next successor of n (i.e., go to line 4)
13  |   |   end
14  |   |   // If found a solution below the bound - return it
15  |   |   if n' is a goal node then
16  |   |   |   return the best path to the goal
17  |   |   end
18  |   |   // Add n' to OPEN or update its location
19  |   |   if n' is in OPEN then
20  |   |   |   Update the key of n' in OPEN using the new g(n')
21  |   |   else
22  |   |   |   Insert n' to OPEN
23  |   |   end
24  |   end
25  end
26  return None // no solution exists that is under the cost bound C
```

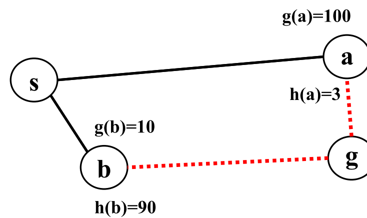


Fig. 1. An example of an expansion dilemma.

To motivate the concept of the potential of a node, consider the graph presented in Fig. 1. Assume that we are searching for a path from s to g. After expanding s, the search algorithm needs to decide which node to expand next: node a or node b. If the task is to find an optimal path from s to g, then clearly b should be expanded first, since there may be a path from s to g that passes through b whose cost is lower than the cost of the path that passes through a (as $f(b) = g(b) + h(b) = 100 < f(a) = g(a) + h(a) = 103$).

If, however, the task is to find a path of cost less than 120 ($C = 120$), then expanding b is not necessarily the best option. For example, it might be better to expand a, which is probably very close to a goal of cost less than 120 (as $h(a) = 3$). Informally, we may say that a has more *potential* to lead to a solution of cost lower than C than b. Next, we define the notion of potential, and show its relation to $u(\cdot)$.

The *potential* of a node n is defined as the probability that a node with h value equal to $h(n)$ has a path to a goal with a cost smaller than $C - g(n)$. For clarity, we omit here the mathematical prerequisites for defining probabilities in this context and provide them later (see Definition 4).

Definition 2 *(Potential).* Given a heuristic function h and a cost bound C, we define the *potential* of node n, denoted $PT_{h,C}(n)$, as

$$PT_{h,C}(n) = \Pr\big(g(n) + h^*(n) < C \,\big|\, h(n)\big)$$

Clearly, the potential of a node depends on the relation between the heuristic function that is used ($h(\cdot)$) and the value it estimates ($h^*(\cdot)$). In some domains, the relation between h and h^* is a known property of h (e.g., a precision parameter of a sensor). In other domains, it is possible to evaluate how close h is to h^* using attributes of the domain. In general, one would expect $h(n)$ to be closer to $h^*(n)$ if $h(n)$ is small. For example, consider a shortest path problem in a map, using the air distance as a heuristic. If the air distance between two nodes is very large, it is more likely that obstacles exist between them. More obstacles imply a greater difference between the air distance and the real shortest path.

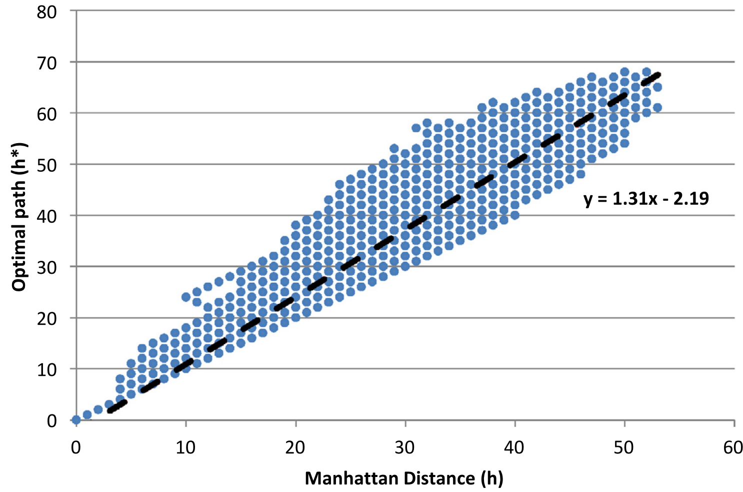


Fig. 2. MD heuristic vs. true distance for the 15-puzzle domain.

Consider the case where for a node n there is a linear stochastic relation between $h(n)$ and $h^*(n)$, i.e., $h^*(n) = h(n) \cdot X_n$, where $X_n \sim X$ is a random variable drawn from a probability distribution X. We assume that the distribution X may be unknown, but that all X_n (for every node n) are independent and identically distributed (i.i.d.) according to X. We denote these assumptions about the relation between h and h^* as the *linear-relative* assumption.[9] Assuming the linear-relative assumption, the potential of a node n is given by:

$$\Pr\bigl(g(n) + h^*(n) < C \bigm| h(n)\bigr) = \Pr\bigl(g(n) + h(n) \cdot X_n < C\bigr)$$
$$= \Pr\bigl(X_n \cdot h(n) < C - g(n)\bigr) = \Pr\left(X_n < \frac{C - g(n)}{h(n)}\right) = \Pr\bigl(X_n < u(n)\bigr)$$

If we do not know the distribution X that X_n is drawn from, the potential of a node cannot be explicitly calculated. However, given any two nodes n_1 and n_2, we *can* determine which node has a higher potential by comparing their $u(\cdot)$ values. This is true because X_{n_1} and X_{n_2} are i.i.d. ($X_{n_1}, X_{n_2} \sim X$), and therefore:

$$u(n_1) \geq u(n_2) \quad \Leftrightarrow \quad \Pr\bigl(X_{n_1} < u(n_1)\bigr) \geq \Pr\bigl(X_{n_2} < u(n_2)\bigr)$$

This idea is summarized in Lemma 1 below.

Lemma 1. *Under the linear-relative assumption, for any pair of nodes n_1 and n_2, we have that $u(n_1) \geq u(n_2)$ iff n_1's potential is greater than or equal to n_2's potential.*

Consequently, for any problem where the linear-relative assumption holds, a BFS that expands the node with the highest $u(\cdot)$ in *OPEN* ($\equiv$ PTS) will expand nodes exactly according to their potential. This result is summarized in Theorem 1:

Theorem 1. *Under the linear-relative assumption, PTS always expands the node with the highest potential.*

It might be hard to find a domain where the *linear-relative* assumption holds exactly. However, this assumption holds approximately for many domains. For example, consider the 15-puzzle, a well-known search benchmark. We solved optimally each of the 1000 standard random instances [36]. For each of these instances we considered all the states on the optimal path, to a total of 52 523 states. Each of these states was assigned a 2-dimensional point, where the x value denotes the Manhattan Distance (MD) heuristic of the state and the y value denotes its optimal cost to the goal. The plot of these points is presented in Fig. 2. The dashed line indicates the best linear fit for the plot, which is the line $y = 1.31 \cdot x - 2.19$. It is easy to observe that a linear fit is very close, suggesting that the linear-relative assumption approximately holds in this domain. Appendix A shows two other domains where this also occurs. In Section 6 we generalize PTS to handle cases where the linear-relative assumption does not hold.

[9] This model is reminiscent of the *constant relative error* [35], where $h^*(n) \leq T \cdot h(n)$ for some constant T.

3.5. Limitations of PTS

Later in this paper (Section 5) we provide experimental results on five domains showing that PTS is effective and robust in practice. However, we would also like to point out two limitations of PTS.

3.5.1. Memory requirements

PTS is a best–first search, storing all the nodes visited during the search in *OPEN* and *CLOSED*. Thus, the memory required for PTS to solve a problem is, in the worst case, exponential in the depth of the search. This prevents running PTS to solve problems in large domains such as the 24-puzzle for close to optimal cost bounds. One might consider running an iterative deepening scheme (like IDA* [2]) with the potential utility function $u(\cdot)$. This would be problematic as $u(\cdot)$ is a non-integer number and thus setting the thresholds of the iterative deepening procedure would be non-trivial (although there are methods to cope with that [37,38]).

3.5.2. Distance vs. cost heuristics

In some domains, the cost of reaching the goal and the distance (number of steps) to reach the goal are not correlated [39]. Effective search algorithms for such domains employ two types of heuristics – one that estimates the cost of reaching a goal, denoted by $h(\cdot)$, and one that estimates the number of steps required to reach a goal, denoted by $d(\cdot)$ [20]. The PTS evaluation function $u(\cdot)$ considers only $h(\cdot)$. Thus PTS may be less effective in domains where $d(\cdot)$ and $h(\cdot)$ are very different. Work on combining $h(\cdot)$ and $d(\cdot)$ for bounded-cost search showed promising results [40].

Resolving these two problems is beyond the scope of this paper and remains as a challenge for future work. Next, we describe APTS/ANA*, which is a non-parametric anytime search algorithm using the PTS evaluation function $u(\cdot)$.

4. Anytime Non-Parametric A* and Anytime Potential Search

In this section we present a framework for constructing a non-parametric anytime search algorithm by solving a sequence of bounded-cost search problems. As an instance of this framework, we present Anytime Potential Search (APTS), which uses PTS to solve these bounded-cost search problems. Then, in Section 4.3 we present a different analytical derivation of APTS as a non-parametric modification of the ARA* anytime search algorithm. The result of this derivation is known as Anytime Non-Parametric A* (ANA*) [9]. ANA* and APTS are the same algorithm, which we call APTS/ANA* to reflect its dual origin.

4.1. Rationale for a non-parametric anytime search algorithm

In general, the need for non-parametric algorithms is prevalent in many fields of computer science and AI [41–43, inter alia]. The main reason is that algorithms with parameters require some method of selecting the values for this parameter, i.e., a method for parameter tuning. Parameter tuning are often:

- **Time consuming.** Parameter tuning are commonly done by running the parametric algorithm several times on a range of instances and parameter values.
- **Crucial for a successful implementation.** Different parameter values can have great impact on the performance of parametric algorithms. This is known to occur in parametric search algorithms [5] and we observe this as well in our experimental results (given later in the paper).

As a result, non-parametric algorithms are more accessible to practitioners. Thus, the need for a non-parametric anytime search algorithm is great.

4.2. A greedy non-parametric anytime search

Algorithm 2 presents a high-level view of an anytime search algorithm. It consists of iteratively calling the ImproveSolution procedure (line 4), which searches for a solution that is better than the incumbent solution (named "Incumbent" in Algorithm 2), whose cost is denoted by *cost*(*Incumbent*) and maintained in the variable G. Initially, there is no incumbent solution, so G is set to ∞ (line 1). When there is no solution better than G, then G is proven to be the optimal solution cost and the search can halt (line 9).[10] Alternatively, the search may be halted earlier, in which case *Incumbent* is returned (line 12).

Different anytime algorithms vary by the implementation of the ImproveSolution procedure (line 4). For example, one can apply a depth first search for ImproveSolution. If every call to ImproveSolution resumes the same depth-first search, then this is identical to DFBnB. Many efficient anytime search algorithms, such as AWA* [22], ARA* [6] and RWA* [24], implement ImproveSolution as different variants of WA*. These WA*-based anytime search algorithms require one or more parameters to set the w for WA*'s f_w evaluation function. Tuning these parameters is often non-trivial.

[10] Some anytime search algorithms cannot guarantee that no better solution is found. We do not discuss such algorithms in this paper.

Algorithm 2: General anytime search algorithm.

```
1  G ← ∞
2  Incumbent ← ∅
3  while search not halted do
4  |  NewSolution ← ImproveSolution // Seek a solution of cost < G
5  |  if NewSolution was found then
6  |  |  G ← cost(NewSolution)
7  |  |  Incumbent ← NewSolution
8  |  else
9  |  |  Return Incumbent (which is optimal)
10 |  end
11 end
12 return Incumbent
```

It is sometimes possible to intelligently customize the parameters of an anytime search algorithm. For example, if one does know beforehand exactly when the search will be halted, it is possible to employ Deadline-Aware Search [44], which estimates during the search which paths to the goal will be achievable before the search is halted. Another example is when there exists a known utility tradeoff between computation time and solution quality. In this case, it is possible to use Best–First Utility-Guided Search [45], which tries to optimize this tradeoff.

In this paper we do not assume any prior knowledge on the termination time of the search, nor do we assume a given utility function that trades off computation for solution quality. In the absence of these, we propose the following greedy approach to anytime search. In every call to ImproveSolution, try to find a solution with cost lower than G as fast as possible. It is easy to see that every such call to ImproveSolution is exactly a bounded-cost search problem, where the cost-bound C is set to be the cost of the incumbent solution (G). Thus, we can use PTS in every call to ImproveSolution (line 4) with cost bound G. The resulting anytime search algorithm does not require parameter tuning (e.g., of w) and is shown empirically to be superior to other anytime search algorithms on a wide range of domains (see Section 5). This anytime search algorithm, which uses PTS for its Improve Solution part, is called Anytime Potential Search (APTS). As stated above, we refer to it as APTS/ANA* to emphasize its second derivation, described later in Section 4.3.

Because initially there is no incumbent solution, the purpose of the first iteration of APTS/ANA* is to find a solution as fast as possible, without any bound on its cost. Thus, $C = \infty$ and as a result the PTS evaluation function $u(n) = \frac{C-g(n)}{h(n)}$ is also equal to ∞ for all the nodes in *OPEN*, making them indistinguishable.

However, as C approaches infinity, the node in *OPEN* with the highest $u(n) = \frac{C-g(n)}{h(n)}$ is actually the node with the lowest $h(n)$. Thus, as C approaches infinity, PTS converges to pure heuristic search. Therefore, we define the first ImproveSolution call of APTS/ANA* to run pure heuristic search until the first solution is found.[11]

4.2.1. Reusing information between PTS calls

There are several known approaches for using knowledge from previous calls to ImproveSolution. In some cases it is best to ignore it completely [24] and restart the search from the initial state. In other cases, a more sophisticated mechanism called *repairing* is preferred [6] (this is what gave the Anytime Repairing A* algorithm its name). For a comprehensive survey and empirical evaluation of the different ways to consider knowledge from previous calls to ImproveSolution, see [27]. No approach has dominated the others in all domains. For simplicity, we chose to implement all anytime algorithms in this paper such that *OPEN* is passed between calls to ImproveSolution. However, modifying APTS/ANA* to use any of the other approaches is trivial.

Consequently, when PTS is called with a new cost bound (which is the cost of the solution found by the previous call to PTS), it does not start from the initial node. Instead, PTS will expand nodes from *OPEN* of the previous PTS call. However, this incurs an overhead, as all the nodes in OPEN need to be reordered according to $u(\cdot)$, as the cost of the incumbent solution has changed.

Next, we give a different derivation of APTS/ANA*, as a non-parametric improvement of the ARA* anytime search algorithm.

4.3. APTS/ANA* as an improved Anytime Repairing A*

ARA* [6] is a well-known anytime search algorithm shown to be effective in many domains [27]. For completeness, a simplified version of the ARA* algorithm is listed in Algorithms 3 and 4. ARA* is a special case of the high-level anytime algorithm described in Algorithm 2. It repeatedly calls its version of ImproveSolution, called here ARA*-ImproveSolution, which aims to find a solution that is w-suboptimal. Initially, $w = w_0$, and it is decreased by a fixed amount Δw after every call to ARA*-ImproveSolution (line 11 in Algorithm 3).

[11] An alternative approach is to use the function $u'(n) = 1 - \frac{h(n)}{C-g(n)}$ instead of $u(n)$. It is easy to see it achieves the same node ordering, but there increasing C to infinity results in an elegant convergence to pure heuristic search.

Algorithm 3: Simplified anytime repairing A*(ARA*).

```
Input: s, the initial state
Input: w_0, the initial w
Input: Δw, the amount by which w is decreased in every iteration
1  G ← ∞; w ← w_0; OPEN ← ∅; Incumbent ← ∅
2  Insert s into OPEN
3  while OPEN is not empty do
4  |  NewSolution ← ARA*-ImproveSolution(OPEN, w, G)
5  |  if NewSolution is not None then
6  |  |  G ← cost(NewSolution)
7  |  |  Incumbent ← NewSolution
8  |  else
9  |  |  return Incumbent
10 |  end
11 |  w ← w − Δw
12 |  Update keys in OPEN and prune nodes with g(·) + h(·) ≥ G
13 end
14 return Incumbent
```

Algorithm 4: ARA*-ImproveSolution.

```
Input: OPEN, the open list
Input: w, the weight of h
Input: G, the cost of the incumbent solution
1  while OPEN is not empty do
2  |  n ← argmin_{n∈OPEN} f_w(n)  // pop the node with the lowest g + wh
3  |  if G ≤ f_w(n) then
4  |  |  return None  // G is proven to be w-admissible
5  |  end
6  |  foreach successor n' of n do
7  |  |  if n' ∉ OPEN or g(n) + c(n, n') < g(n') then
8  |  |  |  g(n') ← g(n) + c(n, n')
9  |  |  |  if g(n') + h(n') < G then
10 |  |  |  |  if n' is a goal then return the best path to the goal;
11 |  |  |  |  Insert n' to OPEN (or update its position if n' ∈ OPEN)
12 |  |  |  end
13 |  |  end
14 |  end
15 end
16 return None  // no solution better than G exists
```

ARA*-ImproveSolution, listed in Algorithm 4, is a variant of WA*, expanding the node $n \in OPEN$ with minimal $f_w(n) = g(n) + w \cdot h(n)$. It terminates either when an improved solution is found (line 10 in Algorithm 4), or when $G \le \min_{n\in OPEN}\{f_w(n)\}$ (line 4 in Algorithm 4), in which case the incumbent solution is proven to be w-suboptimal [6]. An open challenge that we address next is how to set the value of ARA*'s parameters (w_0 and Δw) intelligently.

A property of a good anytime algorithm is that it finds an initial solution as soon as possible, so that a solution can be returned even if little time is available. In general, the higher w is, the greedier the search is and the sooner a solution will be found. Therefore, ideally, w_0 would be set to ∞. However, setting $w_0 = \infty$ is not possible in ARA*, as w is later decreased with finite steps (line 11 in Algorithm 3). For that reason, in ARA* w is initialized with a finite value w_0.

A second desirable property of an anytime algorithm is to reduce the time spent between improvements of the solution, such that when the incumbent solution is requested, the least amount of time has been spent in vain. The amount Δw by which w is decreased should therefore be as small as possible (this is also argued in [6]). However, if w is decreased by too little, the subsequent iteration of ARA*-ImproveSolution might not expand a single node. This is because ARA*-ImproveSolution returns when $\min_{n\in OPEN}\{f_w(n)\} > G$ (line 4 in Algorithm 4). If w is hardly decreased in the next iteration, it might still be the case that $\min_{n\in OPEN}\{f_w(n)\} > G$.

So, what is the maximal value of w for which at least one node can be expanded? That is when

$$w = \max_{n\in OPEN}\left\{\frac{G-g(n)}{h(n)}\right\} = \max_{n\in OPEN}\left\{u(n)\right\} \tag{1}$$

which follows from the fact that

$$f_w(n) \le G \iff g(n) + w \cdot h(n) \le G \iff w \le \frac{G-g(n)}{h(n)} = u(n)$$

The one node that can then be expanded is indeed the node $n \in OPEN$ with a maximal value of $u(n)$. This is precisely the node that APTS/ANA* expands.

One could imagine an adapted version of ARA* that uses Eq. (1) to set w after each iteration of ARA*-ImproveSolution. This would also allow initializing w to ∞ for the first iteration, as is done in APTS/ANA*. However, even such a variation of ARA* is not maximally greedy to find an improved solution, as explained next. Assume that ARA*-ImproveSolution is called with $w = \max_{n \in OPEN}\{u(n)\}$, and let $\hat{w}$ denote this specific value of w. In ARA*, the value of w is fixed throughout a call to ARA*-ImproveSolution. However, during an ARA*-ImproveSolution call a node n might be generated for which $f_{\hat{w}}(n) < G$. If $f_{\hat{w}}(n) < G$, then node n could have been expanded if w was increased even further (up to $u(n)$). A higher w corresponds to a greedier search, so instead one could always maximize w such that there is at least one node $n \in OPEN$ for which $f_w(n) \le G$. This is equivalent to what APTS/ANA* does, by continually expanding the node $n \in OPEN$ with a maximal value of $u(n)$. Thus, APTS/ANA* can be derived as non-parametric tuning of w, such that the search is as greedy as possible, but still has nodes that can be expanded with f_w smaller than G.

Additionally, every time w is changed in ARA*, all the nodes in *OPEN* must be reordered to account for the new w value (line 12 in Algorithm 3). w is updated not only when a new solution is found, but also when the incumbent solution is proven to be w-suboptimal (line 4 in Algorithm 4). Reordering all the nodes in *OPEN* takes $O(|OPEN|)$ time, which can be very large.[12] Thus, an additional benefit of APTS/ANA* over ARA* is that *OPEN* needs to be reordered only when a new solution is found.

*4.4. Suboptimality bounds of APTS/ANA**

Consider the case where h is an admissible heuristic. In this case there is a strong relation between the u-value of the node expanded by APTS/ANA* and the suboptimality bound of the incumbent solution: each time a node n is selected for expansion by APTS/ANA*, $u(n)$ bounds the suboptimality of the incumbent solution. We prove this theorem below. We use G^* to denote the cost of an optimal solution, so $\frac{G}{G^*}$ is the true suboptimality of the incumbent solution. $g^*(n)$ denotes the cost of an optimal path between the start state and n.

Theorem 2. *If $h(n)$ is admissible then $\max_{n \in OPEN}\{u(n)\} \ge \frac{G}{G^*}$. In other words, if a node n is selected by APTS/ANA* for expansion, then $u(n)$ is an upper bound on the suboptimality of the current solution.*

Proof. APTS/ANA* prunes any node n with $g(n) + h(n) \ge G$. Thus, for every node n in *OPEN* it holds that:

$$g(n) + h(n) < G \quad \Rightarrow \quad 1 < \frac{G - g(n)}{h(n)} = u(n)$$

Hence, all the nodes in *OPEN* have $u(n) \ge 1$. Thus, Theorem 2 holds trivially if the current solution is optimal. If an optimal solution has not yet been found, there must be a node $n' \in OPEN$ that is on an optimal path to a goal and whose g value is optimal, i.e., $g(n') = g^*(n')$ (see Lemma 1 in [1]). The minimal cost to move from n' to the goal is $G^* - g^*(n')$, since n' is on an optimal path to a goal. As the heuristic is admissible, $h(n') \le G^* - g^*(n')$. Therefore:

$$u(n') = \frac{G - g(n')}{h(n')} = \frac{G - g^*(n')}{h(n')} \ge \frac{G - g^*(n')}{G^* - g^*(n')} \ge \frac{G}{G^*}$$

where the last inequality follows as $G > G^* \ge g^*(n') \ge 0$. As a result,

$$\max_{n \in OPEN}\{u(n)\} \ge u(n') \ge \frac{G}{G^*} \qquad \square$$

Theorem 2 provides an interesting view on how APTS/ANA* behaves. The suboptimality bound is given by the maximum value of $u(\cdot)$, and APTS/ANA* always expands that node. Thus, APTS/ANA* can be viewed as an informed effort to gradually decrease the suboptimality bound of the incumbent solution. Of course, the children of the expanded node n may have a larger $u(\cdot)$ value than n, in which case $\max_{n \in OPEN}\{u(n)\}$ may even increase, resulting in a worse bound than before the node expansion. This can be overcome by maintaining the best (i.e., the lowest) suboptimality bound seen so far.

Note that this suboptimality bound is available when running APTS/ANA* with almost no additional overhead. Previous approaches to provide a suboptimality bound to an anytime search algorithm used $f_{min} = \min_{n \in OPEN} g(n) + h(n)$. Given f_{min} the suboptimality of the incumbent solution is at most $\frac{G}{f_{min}}$. While this bound can be shown to be tighter than the bound provided by $u(\cdot)$, calculating it required maintaining f_{min}, which requires an additional priority queue that is ordered by $g(n) + h(n)$, while APTS/ANA* uses only a single priority queue.[13]

[12] In fact, reordering *OPEN* requires $O(|OPEN| \log |OPEN|)$ in general. However, it might be possible to use bucket sort to achieve an $O(|OPEN|)$ runtime.

[13] Note that maintaining f_{min} requires more than simply storing the lowest f value seen in the search, since f_{min} increases during the search when the node with f value of f_{min} is expanded. This requires going over OPEN to find the node with the minimal f value, or maintaining an additional priority queue as mentioned above.

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

1	5	2	3
4		6	7
8	9	10	11
12	13	14	15

Fig. 3. The 15-puzzle goal state (left) and a state two moves from the goal (right).

In summary, APTS/ANA* improves on ARA* in five ways: (1) APTS/ANA* does not require parameters to be set; (2) APTS/ANA* is maximally greedy to find an initial solution; (3) APTS/ANA* is maximally greedy to improve the incumbent solution; (4) APTS/ANA* only needs to update the keys of the nodes in *OPEN* when an improved solution is found; and (5) APTS/ANA* makes an informed effort to gradually decrease the suboptimality bound.

4.5. Limitations of APTS/ANA*

While APTS/ANA* has all the attractive properties listed above, it also has some limitations. Since APTS/ANA* runs a sequence of PTS calls, all the limitations described for PTS in Section 3.5 apply also to APTS/ANA*. In addition, APTS/ANA* has two further limitations:

4.5.1. Finding the initial solution

If the heuristic is inaccurate and there are not many goals, finding even a single, unbounded solution can be hard. As defined above, until the first solution is found, APTS/ANA* runs a pure heuristic search. Thus, APTS/ANA* would be inefficient in domains where pure heuristic search is inefficient. For example, in domains where there is a distance-to-go heuristic ($d(n)$), a much more effective way to find a solution fast is to search according to $d(n)$ rather than $h(n)$ [46,39].

4.5.2. Finding too many solutions

Every time a better solution is found, APTS/ANA* is required to re-sort all the nodes in *OPEN* (Algorithm 2) to account for the new incumbent solution (the $u(\cdot)$ values change). Thus, if there are many solutions, each slightly better than the previous, then APTS/ANA* would suffer from the overhead of re-sorting *OPEN* every time a new, better, solution is found.

There are ad hoc solutions to these limitations. Instead of using APTS/ANA* to find the first solution, it is possible to use another algorithm (e.g., Speedy search [46]) to find the first solution and provide it to APTS/ANA* as an initial incumbent solution. Additionally, the bound of the next PTS can be set to be lower than the incumbent solution by some Δ, to avoid finding too many solutions. Such a Δ, however, would be a parameter of the algorithm and setting it raises the parameter tuning problem we aim to avoid.

5. Experimental results

In this section we empirically evaluate the performance of PTS as a bounded-cost search algorithm, and APTS/ANA* as an anytime search algorithm. This is done over a range of domains: 15-puzzle, KPP-COM, robot arm, grid world planning and multiple sequence alignment (MSA). Next, we describe the domains used in our experiments.

5.1. Domains

For every domain, we provide the domain details (what is a state etc.), the heuristic function used, and how the problem instances were generated.

5.1.1. The 15-puzzle

The 15-puzzle is a very well-known puzzle that consists of 15 numbered tiles that can be moved in a 4×4 grid. There is one blank location in the grid. The blank can be swapped with an adjacent tile. The left part of Fig. 3 shows the goal state that we used for the 15-puzzle while the right part shows a state created from the goal state by applying two operators, namely swapping the blank with tile 1 and then swapping it with tile 5. The number of states reachable from any given state is $(4^2)!/2$ [47]. The task is to find a short path from a given starting state to the goal state. The 15-puzzle is a common search benchmark [2,15,36,48,49].

There are many advanced heuristics for the 15-puzzle. In the experiments below we chose the simple Manhattan Distance heuristic (MD) because our goal is to compare search algorithms and not different heuristics. The experiments were performed on Korf's standard 100 random 15-puzzle instances [2].

5.1.2. Key Player Problem in Communication (KPP-COM)

The Key Player Problem in Communication (KPP-COM) is the problem of finding a set of k nodes in a graph with the highest Group Betweenness Centrality (GBC). GBC is a metric for centrality of a group of nodes [50]. It is a generalization of the *betweenness* metric, which measures the centrality of a node with respect to the number of shortest paths that pass through it [51]. Formally, the betweenness of a node n is $C_b(n) = \sum_{s,t \in V s,t \neq n} \frac{\sigma_{st}(n)}{\sigma_{st}}$, where σ_{st} is the number of shortest

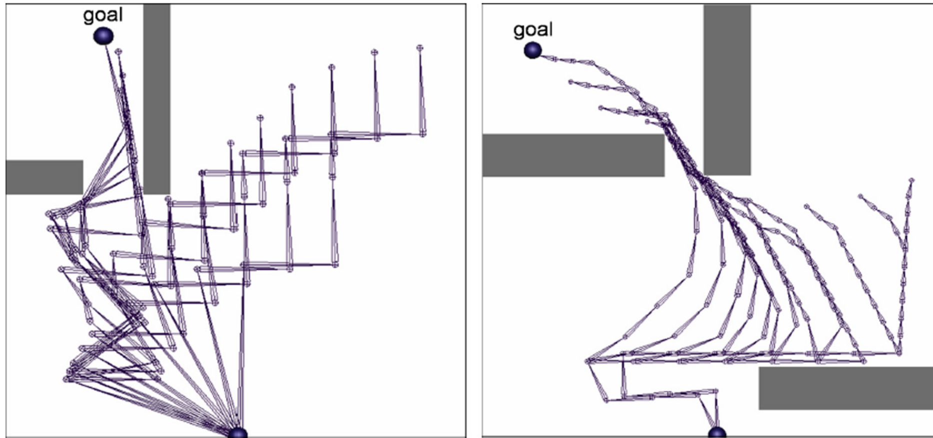


Fig. 4. Example of the motion of a 6 DoF (left) and 20 DoF (right) robot arm [6].

paths between s and t and $\sigma_{st}(n)$ is the number of shortest paths between s and t that pass through n. The betweenness of a group of nodes A, termed group betweenness, is defined as $C_b(A) = \sum_{s,t \in V \setminus A} \frac{\sigma_{st}(A)}{\sigma_{st}}$, where $\sigma_{st}(A)$ is the number of shortest paths between s and t that pass through at least one of the nodes in A.

KPP-COM is known to be NP-Hard [52]. It has important network security applications, such as optimizing the deployment of intrusion detection devices [53]. KPP-COM can be solved as a search problem. Let $G = (V, E)$ be the input graph in which we are searching for a group of k vertices with the highest GBC. A state in the search space consists of a set of vertices $N \subseteq V$, $|N| \leq k$. N is considered as a candidate to be the group of vertices with the highest GBC. The initial state of the search is an empty set, and each child of a state corresponds to adding a single vertex to the set of vertices of the parent state. Every state has a value, which is the GBC of the set of vertices it contains. While the goal in the previous domain (the 15-puzzle) is to find a solution of *minimal* cost, the goal in KPP-COM is to find a solution with *maximum* value. We call problems of the former type *MAX problems* and problems of the latter type *MIN problems*. An admissible heuristic for MAX problems is required to be an *upper bound* on the optimal value. Similarly, a suboptimal solution is one with a smaller value than the optimal solution.

A number of efficient admissible (overestimating) heuristics for this problem exist [52] and in our experiments we used the best one, calculated as follows. Consider a state consisting of a set of m vertices V_m. First, the contribution of every individual vertex $v \in V \setminus V_m$ is calculated. This is the $C_b(V_m \cup \{v\}) - C_b(V_m)$. Then, the contribution of the topmost $k - m$ vertices is summed and used as an admissible heuristic, where k is the total number of vertices that needs to be selected (see [52] for a more detailed discussion about this heuristic).

Since the main motivation for the KPP-COM problem is in communication network domains, all our experiments were performed on graphs generated by the Barabási–Albert model [54]. This random graph model is a well-used model of Internet topology and the web graph, which accepts two parameters: the number of vertices in the graph and a density factor. We have experimented with a variety of graph sizes and density factor values, and present the average results over 25 graphs with 600 vertices with a density factor of 2. Additionally, we have limited the size of the searched group of vertices to be 20 (i.e., $k = 20$).

5.1.3. Robot arm

The robot arm domain is taken from Maxim Likhachev's publicly available SBPL library (http://www.sbpl.net). This problem illustrates the performance of the proposed algorithms in a domain with a high branching factor and few duplicate states.

We consider both a 6 degrees of freedom (DoF) arm and a 20 DoF arm with a fixed base in a 2D environment with obstacles, shown in Fig. 4. The objective is to move the end-effector from its initial location to a goal location while avoiding obstacles. An action is defined as a change of the global angle of any particular joint (i.e., the angle with respect to a fixed initial point) having the next joint further along the arm rotate in the opposite direction to maintain the global angle of the remaining joints. All actions have the same cost.

The environment is discretized into a 50×50 2D grid. The heuristic is calculated as the shortest distance from the current location of the end-effector to the goal location that avoids obstacles. To avoid having the heuristic overestimate true costs, joint angles are discretized so as to never move the end-effector by more than one cell on the 50×50 grid in a single action. Note that the size of the state space for this domain is 10^9 states for the 6 DoF robot arm and more than 10^{26} states for the 20 DoF robot arm.

5.1.4. Gridworld planning

The next domain we considered is that of two planar gridworld path-planning problems with different sizes and numbers of obstacles, also taken from Likhachev's SBPL library. This problem illustrates the performance of the algorithms in a domain

Fig. 5. An example of aligning 8 DNA sequences.

Table 1
Competing algorithms for each domain.

Algorithms	KPP-COM	15-Puzzle	Robot arm	Gridworld	MSA
AWA*	√	√	–	–	–
ARA*	–	–	√	√	√
DFBnB	√	–	–	–	–

with many transpositions and a relatively small branching factor. We set the start state as the cell at the top left corner, and the goal state at the bottom right cell. The first gridworld problem is a 100×1200 8-connected grid with obstacles, with unit cost to move between adjacent obstacle-free cells. The second gridworld problem is a 5000×5000 4-connected grid in which each transition between adjacent cells is assigned a random cost between 1 and 1000. For the 5000×5000 4-connected grid problem with non-unit edge cost, we considered two cases, one with obstacles and one without.

5.1.5. Multiple sequence alignment (MSA)

Our final domain is the MSA problem. The uniqueness of this domain (in comparison to the domains above) is its high branching factor and non-uniform edge costs (i.e., costs of different edges may vary).

MSA is a central problem for computational molecular biology when attempting to measure similarity between a set of sequences [55]. The input to MSA is a set of sequences of items (e.g., gene or protein sequences). Every sequence is mapped to an array, while possibly leaving empty spaces. This mapping is referred to as an *alignment*. Fig. 5 shows an example of an alignment of 8 sequences. Alignments have a cost, according to a biologically motivated interpretation of the alignment. We used the sum-of-pairs cost function, used by many previous works [56,57,55, inter alia]. According to this cost function the alignment cost of k sequences is the sum of the alignment costs of all $\binom{k}{2}$ pairs of sequences. The alignment cost of a pair of sequences x and y considers the number of *matches*, *gaps* and *substitutions* in the alignment. A match occurs when two identical items (e.g., the same protein) are mapped to the same array index. A gap occurs when only a single sequence is using an index. A *substitution* occurs when different items are mapped to the same index. The alignment cost we used assigned to every match, gap and substitution a cost of zero, one and two, respectively.

MSA can be formalized as a shortest-path problem in an n-dimensional lattice, where n is the number of sequences to be aligned [58]. A state is a (possibly partial) alignment represented by the items aligned so far from each sequence. A goal is reached when all the characters in all the sequences are aligned. A move in the search space assigns a specific index to an item in one or more sequences. The cost of a move is computed as the cost of the partial alignment. The MSA problem instances we experimented with consist of five dissimilar protein sequences obtained from [59] and [60]. The heuristic is based on summing the optimal pairwise alignments, which were precomputed by dynamic programming [60].

5.2. Bounded-cost experiments

In this section we empirically evaluate the performance of PTS. In every experiment, the cost bound C was set, and PTS and the competing algorithms were run until a solution with cost lower than C was found. This was repeated for a range of cost bounds (i.e., a range of C values). In domains where instances could be solved optimally in reasonable time, we chose the range of C values to cover bounds close to the average optimal solution as well as substantially higher bounds. In domains where solving instances optimally was not feasible, we chose the cost bounds that were solvable by the compared algorithms in reasonable time. The performance of the various algorithms was measured in runtime.

We compared PTS against a range of existing anytime search algorithms: AWA* [22], ARA* [6] and DFBnB [23]. To make the comparison to PTS fair, we also modified these anytime algorithms to prune every node n with $g(n) + h(n)$ larger than the cost bound, as is done for PTS (line 12 in Algorithm 1). Thus, DFBnB can also be viewed as a depth-first search with node pruning, where nodes with $g(n) + h(n)$ larger than the cost bound are pruned.

Different anytime search algorithms performed differently, and in the rest of this section we only show results of the best performing anytime search algorithms for every domain. Table 1 lists the best performing algorithms for every domain. For KPP-COM, these were AWA* and DFBnB. The effectiveness of DFBnB in KPP-COM was not surprising, as DFBnB is

Table 2
15-puzzle bounded-cost results. The % values are the average % of nodes expanded by A*. The values in brackets are runtimes in milliseconds.

C	PTS		AWA*-1.5		AWA*-2.0		AWA*-2.5		AWA*-3.0	
55	%36	(955)	**%21**	(**482**)	%48	(751)	%86	(1068)	%124	(1651)
60	%**11**	(**389**)	%12	(**389**)	%**11**	(418)	%33	(643)	%66	(1144)
65	%**7**	(370)	%12	(389)	%**7**	(**362**)	%13	(430)	%67	(735)
70	%**5**	(**353**)	%12	(388)	%6	(354)	%6	(360)	%17	(464)
75	%**5**	(351)	%12	(387)	%6	(357)	%**5**	(**347**)	%8	(385)
80	%**4**	(**346**)	%12	(389)	%6	(355)	%5	(348)	%**4**	(350)
85	%**4**	(**342**)	%12	(387)	%6	(357)	%5	(347)	%**4**	(343)
90	%**4**	(345)	%12	(392)	%6	(360)	%5	(347)	%**4**	(**340**)

Table 3
KPP-COM bounded-cost results. Values are the average runtime in seconds.

C	320 000	310 000	300 000	290 000	280 000
PTS	1194	**727**	**556**	**408**	**329**
DFBnB	**1117**	735	562	422	347
AWA*-0.7	1138	751	569	424	336
AWA*-0.8	1391	972	758	580	438
AWA*-0.9	2713	1850	1528	1155	796
A*	7229	4389	4118	2729	1750

known to be effective when the depth of the solution is known in advance (in KPP-COM this is k, the size of the searched group) [61,62]. This was also confirmed in our KPP-COM experiments and in previous work [52]. DFBnB is less effective in other domains, where the depth of the search tree varies and there are many cycles. The relative performance of AWA* and ARA* varies between domains. Our experiments showed that AWA* outperformed ARA* in the 15-puzzle (previous work showed this in the 8-puzzle domain [22]), while ARA* performed better in all other domains.[14] Both ARA* and AWA* are parametric algorithms. We experimented with a range of parameters and show the results of the best performing parameter settings we found.

Table 2 displays the results for the 15-puzzle domain. Results in this domain are traditionally shown in terms of nodes expanded, to avoid comparing implementation details. Therefore, in addition to comparing the runtime of each algorithm we also measured the number of nodes expanded. The number of nodes expanded is given in Table 2 as the percentage of nodes expanded with respect to the number of nodes expanded by A* until an optimal solution is found. Runtime is shown in the same table in brackets, and measured in milliseconds. The best performing algorithm for every cost bound is marked in bold.

Results clearly show a drastic reduction in the size of the searched state space (yielding a substantial speedup) over trying to find an optimal solution. For example, PTS can find a solution under a cost bound of 70 by expanding on average only 5% of the nodes that will be expanded by A* to find an optimal solution.

In addition, we see that PTS performs best for the vast majority of cost bounds. PTS also performs relatively well even for the few cost bounds where it is not the best performing algorithm. We can also see the large impact of the w parameter on the performance of AWA*. For example, for cost bound 65, AWA* with $w = 3$ performed more than 9 times worse than AWA* with $w = 2$. This emphasizes the robustness of PTS, which does not require any parameter.

Due to implementation details, there was not a perfect correlation between the number of nodes expanded by an algorithm and its runtime, as part of the runtime is used to allocate memory for the required data structures and other algorithm initialization processes. Nonetheless, similar trends as those noted above can also be seen for runtime (these are the values in brackets in Table 2).

Deeper inspection of the results shows that for the lowest cost bound (55), PTS is outperformed by AWA* with $w = 1.5$. The average cost of an optimal solution for the 15-puzzle instances we experimented with is 52. This suggests that PTS might be less effective for cost bounds that are close to the optimal solution cost. In such cases, a more conservative A*-like behavior, exhibited by AWA* with w close to one, might be beneficial, as A* is tuned to find an optimal solution. We observed a similar trend in some of the other domains we experimented with.

Next, consider the results for KPP-COM, shown in Table 3. Values are runtime in seconds until a solution below the cost bound C was found. Similar trends are observed in this domain: PTS performs best for most cost bounds and the w parameter greatly influences the performance of AWA*.

The differences in performance of PTS, AWA*-0.7 and DFBnB are relatively small in this domain. This is because DFBnB is known to be very efficient in this domain [52]. Hence, PTS could not substantially improve over DFBnB. As for AWA*-0.7, consider the case where w reaches zero. In such a case, AWA* behaves like uniform cost search, a best–first search using only g to evaluate nodes. In MAX problems with non-negative edge costs, as is the case for KPP-COM, a uniform cost search

[14] For some of the results of the other algorithms, see http://goldberg.berkeley.edu/ana/ANA-techReport-v7.pdf.

Table 4
Robot arm bounded-cost results. Values are the average runtime in seconds.

	(a) 6 DoF Robot Arm					(b) 20 DoF Robot Arm				
	Cost bounds					Cost bounds				
C	61	63	65	67	69	C	80	82	85	88
PTS	2764	**36**	**31**	**18**	**14**	PTS	101	84	**16**	**8**
ARA*	**647**	95	61	61	31	ARA*	**40**	**21**	21	21

Table 5
Gridworld planning results. Values are the average runtime in seconds.

(a) 100 × 1200, unit edge costs			
C	1050	1055	1060
PTS	**4.8**	**4.5**	**3.7**
ARA*	478.0	478.0	8.1

(b) 5000 × 5000, with obstacles					
C	4400	4410	4430	4450	4470
PTS	**1.50**	**1.40**	**1.39**	**0.68**	**0.59**
ARA*	36.20	20.10	1.82	1.38	1.38

(c) 5000 × 5000, without obstacles						
C	2550	2600	2700	2800	2900	3000
PTS	**15.630**	**4.878**	**0.496**	**0.265**	**0.093**	**0.072**
ARA*	129.500	8.955	5.476	4.649	3.384	3.167

Table 6
Multiple sequence alignment results. Values are the average runtime in seconds.

C	1585	1590	1595	1600	1605	1610
PTS	**108.7**	**69.0**	**26.4**	**7.4**	**0.7**	0.1
ARA*	789.4	736.1	635.3	427.4	186.9	**0.0**

behaves like DFBnB. Thus, decreasing w towards zero results in AWA* behaving more and more like DFBnB, until eventually converging to DFBnB when $w = 0$. This explains the similar performance of AWA*-0.7 and DFBnB.

Results for the robot arm, gridworld planning, and multiple sequence alignment domains are shown in Tables 4, 5 and 6, respectively. PTS's favorable behavior can be viewed across all domains. For example, in the 100 × 1200 gridworld planning domain, PTS is two orders of magnitude faster than ARA* for a cost bound of 1050. Following all the bounded-cost results shown above, we conclude that PTS outperforms the best performing anytime search algorithm for almost all cost bounds and across the entire range of domains we used. Furthermore, PTS exhibits the most robust performance, without any parameter tuning required.

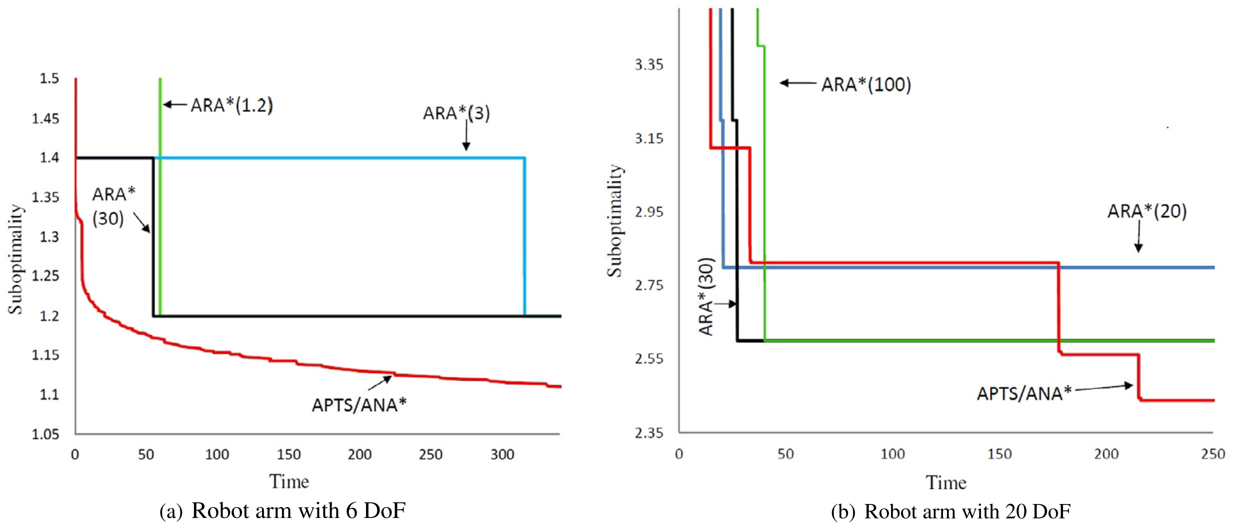
5.2.1. *The effect of w in MAX problems*

The KPP-COM results highlight an interesting question – how does the w parameter affect the performance of AWA* for MAX problems? The best performing instance of AWA* in our KPP-COM experiments used $w = 0.7$, the lowest w we experimented with. Thus, one might think that decreasing w has an equivalent effect to increasing w in MIN problems. Indeed, decreasing w for $w > 1$ in MAX problems is similar to increasing w for $w < 1$ in MIN problems – making an admissible heuristic more informed and the search more efficient. In the limit, however, decreasing w in MAX problems and increasing w in MIN problems have completely different effects. As mentioned above, for MAX problems setting $w = 0$ results in AWA* behaving like uniform cost search. For MIN problems, $w = \infty$ results in AWA* behaving like pure heuristic search. Pure heuristic search is expected to find solutions fast but of low quality,[15] while uniform cost search ignores the heuristic completely and is therefore expected to be slower than a search that uses a heuristic. Exploring the effect of w in MAX problems is left for future work.

5.3. *Anytime experiments*

Next, we present experimental results to evaluate the performance of APTS/ANA* on the same range of domains described in Section 5.1. In every experiment, we ran APTS/ANA* and the best-performing anytime search algorithms (different for each domain), and recorded the suboptimality of the incumbent solution as a function of the runtime. In the following figures, ARA*(X) will denote ARA* using $w_0 = X$ (w_0 is the initial value of w used by ARA*; see Section 2.3).

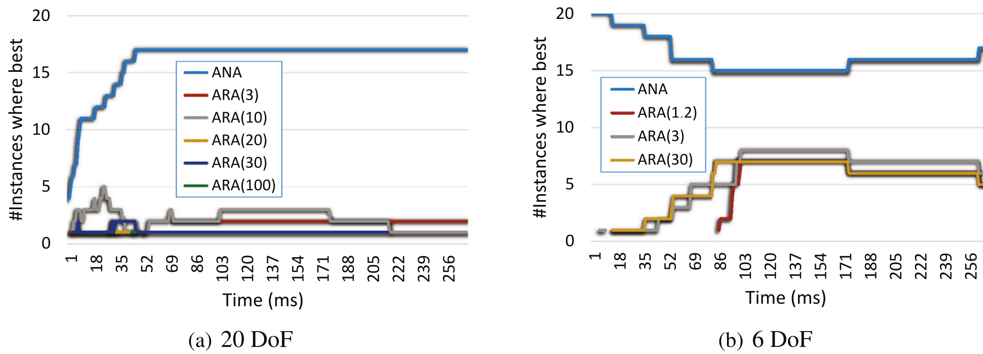
[15] This is a general guideline [6], but deeper study of the effect of w in MIN problems is needed [39].



(a) Robot arm with 6 DoF

(b) Robot arm with 20 DoF

Fig. 6. Anytime experiments for the robot arm domain. Time (x-axis) is in seconds.



(a) 20 DoF

(b) 6 DoF

Fig. 7. # instances where each algorithm found the best (lowest cost) solution.

First, consider the results for the 6 DoF robot arm experiments, shown in Fig. 6(a). The y-axis shows the reported suboptimality, where 1 corresponds to a solution that is optimal. The x-axis shows the runtime in seconds. The benefits of APTS/ANA* in this domain are very clear. APTS/ANA* dominates all other algorithms throughout the search. Although not visible in the figure, we also report that APTS/ANA* found an initial solution faster than ARA*.

Next, consider the performance of APTS/ANA* in the 20 DoF robot arm experiments, shown in Fig. 6(b). In this domain APTS/ANA* is not always the best performing algorithm. For some weights and time ranges, ARA* is able to find solutions of better quality. However, the rapid convergence of APTS/ANA* towards an optimal solution and consistent decrease in suboptimality are very clear in this domain, demonstrating the anytime behavior of APTS/ANA*. As in the 6 DoF experiments, in this domain too APTS/ANA* found an initial solution faster than all other algorithms.

The above robot arm results are for a single but representative instance. We also experimented on a set of randomly generated 6 DoF and 20 DoF robot arm instances (20 instances each) showing, in general, very similar trends. As an aggregated view showing the relative anytime performance of APTS/ANA* with respect to the other algorithms, we also analyzed which was the most effective algorithm throughout the search. This was measured by counting for every algorithm A and every millisecond t, the number of problem instance where the incumbent solution found by A after running t milliseconds is smaller than or equal to the incumbent solution found by all other algorithms after running t milliseconds. This measures the number of instances where algorithm A was the best algorithm if halted after t milliseconds. Fig. 7 shows the results of this analysis, where the x-axis is the runtime (in milliseconds) and the y-axis is the number of instances where each algorithm was best. For both 20 DoF (left) and 6 DoF (right), the results clearly show that APTS/ANA* is the best performing algorithm throughout the search for the majority of problem instances.

The results for the MSA domain, shown in Fig. 8, show that APTS/ANA* (as in the 6 DoF robot arm) dominates all other algorithms throughout the search. For both robot arm and MSA experiments, APTS/ANA* has the largest number of steps in its decrease to a lower suboptimality. Correspondingly, the time between solution improvements is smaller. For example, in the MSA experiments, APTS/ANA* spent an average of 18 s between solution improvements, while in the best run of ARA*,

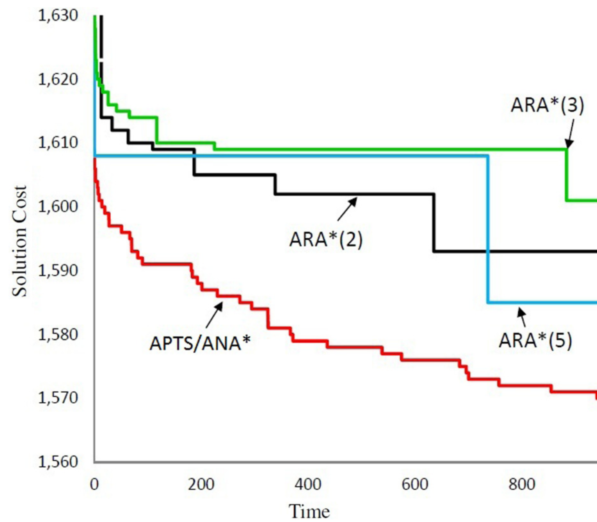


Fig. 8. MSA, 5 sequences. Time (x-axis) is in seconds.

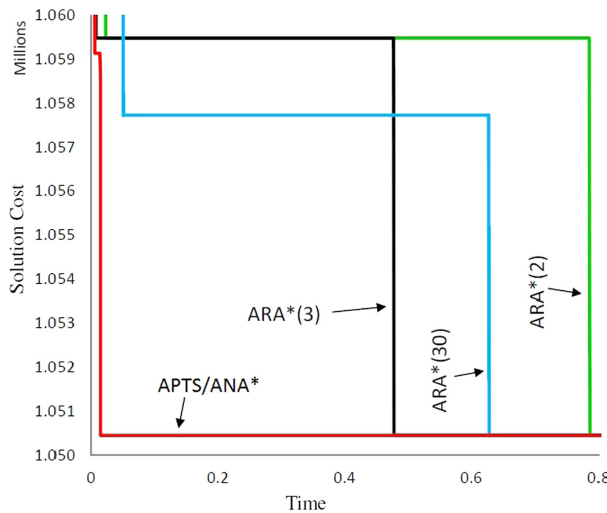


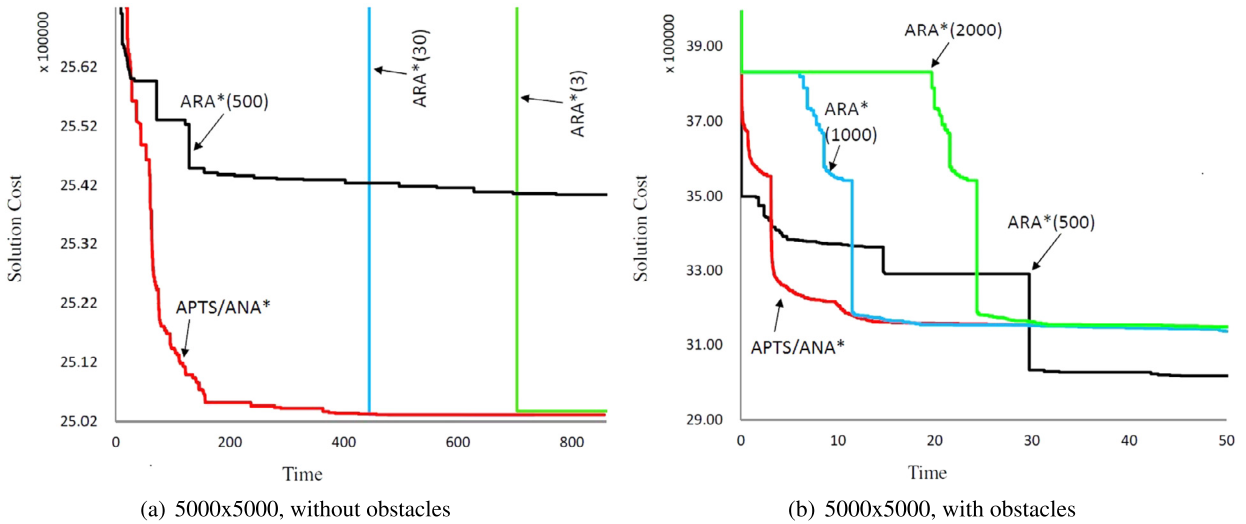
Fig. 9. 100×1200, unit edge cost with obstacles. Time (x-axis) is in seconds.

it took 200 s on average to find a better solution. As discussed in Section 4, this is intuitively a desirable property of an anytime algorithm.

Next, consider the results for the gridworld domain, given in Figs. 9 and 10. For the 100×1200 grid with unit edge cost experiments (Fig. 9) and the 5000×5000 grid with random edge cost experiments (Fig. 10(a)), APTS/ANA* dominates all other algorithms throughout the search. However, the results for the 5000×5000 domain with obstacles (Fig. 10(b)) show a different behavior: after 30 seconds of runtime, ARA* with $w_0 = 500$ returned solutions of lower cost than the solutions returned by APTS. While not shown in the figure, we report that APTS/ANA* required an additional 193 s to find a solution of the same cost as this ARA* instance.

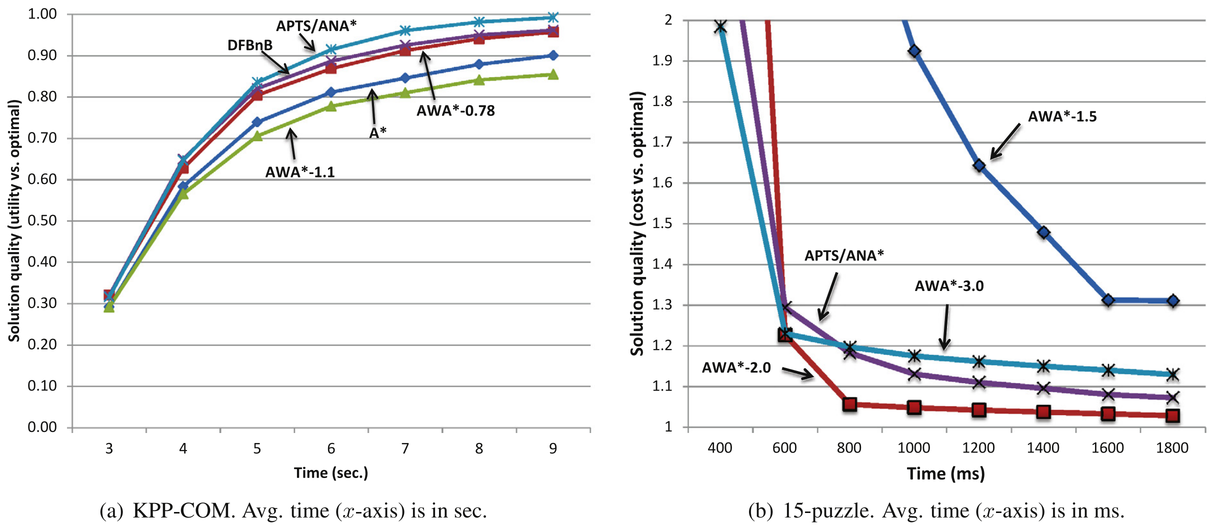
Note that the performance of ARA* varied greatly with the value of w_0 in all the domains we experimented with. For the 5000×5000 gridworld with no obstacles, $w_0 = 30$ is the best w_0 value of those tested, while for the 5000×5000 gridworld with obstacles $w_0 = 500$ is the best w_0 value. The challenge of tuning w_0 in ARA* is especially evident in the 100×1200 gridworld experiment (seen in Fig. 9). The disparity in the ARA* results for different values of w_0 illustrates a non-linear relationship between w_0 and ARA*'s performance. This emphasizes that setting this value is non-trivial, as a higher or lower w_0 value does not necessarily guarantee better or worse performance. Thus, even if for some problem instances ARA* with some parameter settings outperforms APTS/ANA*, APTS/ANA* still has the benefit of not depending on any parameter.

Next, consider the results for the KPP-COM domain, seen in Fig. 11(a). The y-axis denotes the suboptimality of the incumbent solution as a function of the computation time, shown on the x-axis. Since KPP-COM is a MAX problem, the suboptimality starts very close to zero and converges to one from below when an optimal solution is found.



(a) 5000x5000, without obstacles (b) 5000x5000, with obstacles

Fig. 10. Anytime experiments for 5000 × 5000 gridworld. Time (x-axis) is in seconds.



(a) KPP-COM. Avg. time (x-axis) is in sec. (b) 15-puzzle. Avg. time (x-axis) is in ms.

Fig. 11. Anytime experiments for KPP-COM and 15-puzzle domains.

The results in Fig. 11(a) clearly show that in this domain as well, APTS/ANA* performs consistently no worse and often better than the other algorithms. As discussed in Section 5.2, DFBnB is known to be highly effective in this domain [52]. Indeed, the results of APTS/ANA* and DFBnB are almost identical, with a slight advantage for APTS/ANA* when the runtime was more than 5 seconds.

Negative results for APTS/ANA* were obtained for the 15-puzzle domain, shown in Fig. 11(b). Running APTS/ANA* yielded worse results than AWA* with a tuned w value ($w = 2$). This is because, as explained in Section 4, APTS/ANA* runs pure heuristic search until it finds the initial solution. In many domains, including all those described above, this results in finding an initial solution very fast. With proper tie-breaking, this is also the case for the 15-puzzle domain. The quality of this initial solution, however, is very poor. Thus, APTS/ANA* wastes a significant amount of time improving solutions that have very high suboptimality, until it converges to an optimal solution. By contrast, the initial solution found by AWA* with $w = 2$ (AWA*-2.0 in Fig. 11(b)) has a suboptimality of at most 2, and can be found relatively easily in this domain.

As a partial remedy for this problem, we tried first running WA* with different weights and then giving APTS/ANA* the initial solution, after which it continues to run as usual. Results show that this APTS/ANA* variant performed the same as the best performing anytime search algorithm for this domain.

To summarize, on all the domains except the 15-puzzle, APTS/ANA* was competitive, and in most cases significantly better than existing anytime search algorithms in that: (1) APTS/ANA* finds an initial solution faster, (2) APTS/ANA* improves the incumbent solution faster and more often, and (3) APTS/ANA* converges faster to solutions of higher quality. Moreover, in all of the domains we experimented with, we found that the performance of existing parametric anytime search algo-

rithms is greatly affected by the parameter values. By contrast, APTS/ANA* does not require any parameters to be set, and still manages to outperform existing anytime algorithms.

6. Generalized Potential Search (GPTS)

In Section 3 we presented the PTS algorithm, and showed that under the linear-relative assumption PTS always expands the node with the highest potential (Theorem 1). In this section we generalize PTS to cases where this assumption does not hold. We call the resulting algorithm General Potential Search (GPTS) – a search algorithm that always expands the node with the highest potential.

GPTS can be implemented as a BFS with an evaluation function that orders nodes according to their potential (denoted earlier in this paper as $PT_{h,C}$). We call such an evaluation function a Potential Ordering Function (POF).

Definition 3 (*Potential Ordering Function (POF)*). A function F is called a Potential Ordering Function (POF) if for any pair of nodes n_1 and n_2, we have that

$$F(n_1) \geq F(n_2) \quad \text{iff} \quad PT_{h,C}(n_1) \geq PT_{h,C}(n_2)$$

Of course, $PT_{h,C}$ itself is a trivial POF. Calculating $PT_{h,C}$, however, may be non-trivial. Next, we propose practical POFs for a wide range of domains that do not require calculating $PT_{h,C}$ for any node. This suggests the applicability of GPTS.

As a preliminary, we provide a more rigorous definition of the *potential* than given in Definition 2. Let $\mathcal{S}$ be the searched state space, and let $\mathcal{D}$ be a probability distribution over the nodes in $\mathcal{S}$. S denotes a random variable corresponding to a state from $\mathcal{S}$ randomly drawn according to distribution $\mathcal{D}$. We denote by $\Pr(h^*(S) = X \mid h(S) = Y)$ the probability that X is the cost of the lowest cost path to a goal from a random state S with heuristic value Y. The potential of a node n is defined as the probability that a random state S with the same h value as node n will have a path to a goal of cost lower than $C - g(n)$. We next define this formally.

Definition 4 *(Potential).* Given a heuristic function h and a cost bound C, we define the potential of node n, denoted $PT_{h,C}(n)$, as

$$PT_{h,C}(n) = \Pr\big(g(n) + h^*(S) < C \,\big|\, h(S) = h(n)\big)$$

The potential of a node, as defined above, depends on the relation between $h(\cdot)$ and $h^*(\cdot)$. We formalize this h-to-h^* relation by introducing the notion of a *heuristic model*, or h-model in short. An h-model of a heuristic function is a function that captures the probabilistic relation between a heuristic and the lowest cost path to the goal that it estimates.

Definition 5 *(h-Model).* A function $e(\cdot)$ is said to be an h-model of a heuristic function h, if for every h-value v of the heuristic function, and every h^*-value K of a state in the state space,

$$\Pr\big(h^*(S) < K \,\big|\, h(S) = v\big) = \Pr\big(e(v) < K\big)$$

Domains with the linear-relative assumption (described in Section 3) are simply domains with an h-model $e(v) = X \cdot v$, where X is a random independent identically distributed (i.i.d.) variable. Correspondingly, we call this type of h-model the *linear-relative h-model*. Using this notation, Lemma 1 (given in Section 3) states that the PTS evaluation function $u(n) = \frac{C-g(n)}{h(n)}$ is a POF for the linear-relative h-model. Next, we describe POFs for other h-models.

6.1. Additive h-model

Consider the following h-model: $e(v) = v + X$, where X is an i.i.d. random variable. This does not imply that the distribution of X is uniform, but just that the additive error of every node is taken from the same distribution (independently). We call such an h-model an *additive h-model.*[16]

If the distribution of X is known, no non-trivial POF is required, as the potential can be easily calculated:

$$\begin{aligned}
PT_{h,C}(n) &= \Pr\big(g(n) + h^*(S) < C \,\big|\, h(S) = h(n)\big) \\
&= \Pr\big(h^*(S) < C - g(n) \,\big|\, h(S) = h(n)\big) \\
&= \Pr\big(h(n) + X < C - g(n)\big) \\
&= \Pr\big(X < C - g(n) - h(n)\big)
\end{aligned}$$

[16] This is reminiscent of the *bounded constant absolute error* model described by Pearl [35] where the difference between h and h^* is bounded by a constant (i.e., $h^*(n) \leq h(n) + K$). Here, K is the largest value for X.

For example, in the special case where X is a constant K, the potential of a node is a simple binary function:

$$PT_{h,C}(n) = \begin{cases} 1 & g(n)+h(n)+K \le C \\ 0 & \text{otherwise} \end{cases}$$

If the distribution of X is not known, then directly computing $PT_{h,C}$ is impossible. Next, we show that even if the distribution of X is unknown, it is still possible to have a POF. Corollary 1 presents a theoretical POF that applies to any given h-model.

Corollary 1. *$F_C(n) = \Pr(e(h(n)) < C - g(n))$ is a POF.*

Proof. For any pair of nodes n_1, n_2, if $PT_{h,C}(n_1) \ge PT_{h,C}(n_2)$ then:

$$\Pr\big(g(n_1)+h^*(S) < C \,\big|\, h(S)=h(n_1)\big) \ge \Pr\big(g(n_2)+h^*(S) < C \,\big|\, h(S)=h(n_2)\big)$$
$$\Pr\big(h^*(S) < C-g(n_1) \,\big|\, h(S)=h(n_1)\big) \ge \Pr\big(h^*(S) < C-g(n_2) \,\big|\, h(S)=h(n_2)\big)$$
$$\Pr\big(e\big(h(n_1)\big) < C-g(n_1)\big) \ge \Pr\big(e\big(h(n_2)\big) < C-g(n_2)\big) \quad [\text{Definition 5}]$$
$$F_C(n_1) \ge F_C(n_2)$$

Proving the reverse direction is straightforward, and thus F_C is a POF. □

Using the POF given in Corollary 1 requires calculating $F_C(n) = \Pr(e(h(n)) < C - g(n))$. For an additive h-model, $F_C(n)$ still depends on X, since $e(h(n)) = h(n) + X$. Theorem 3 provides an applicable POF that does not depend on X.

Theorem 3. *If the h-model is additive then $-f(n) = -(g(n)+h(n))$ is a POF.*

Proof. For any pair of nodes n_1, n_2, if $PT_{h,C}(n_1) \ge PT_{h,C}(n_2)$ then:

$$\Pr\big(e\big(h(n_1)\big) < C-g(n_1)\big) \ge \Pr\big(e\big(h(n_2)\big) < C-g(n_2)\big) \quad [\text{Corollary 1}]$$
$$\Pr\big(h(n_1)+X < C-g(n_1)\big) \ge \Pr\big(h(n_2)+X < C-g(n_2)\big) \quad [\text{additive } h\text{-model}]$$
$$\Pr\big(X < C-g(n_1)-h(n_1)\big) \ge \Pr\big(X < C-g(n_2)-h(n_2)\big)$$
$$C-g(n_1)-h(n_1) \ge C-g(n_2)-h(n_2) \quad [X \text{ is i.i.d.}]$$
$$-f(n_1) = -g(n_1)-h(n_1) \ge -g(n_2)-h(n_2) = -f(n_2)$$

The reverse direction is straightforward. □

A direct result of Theorem 3 is that for an additive h-model, GPTS and A* expand nodes in the same order.

6.2. General *h*-model

Consider the more general case, where the h-model $e(v)$ is an algebraic combination of v and a random i.i.d. variable X. We overload the function e to describe this algebraic combination, and write $e(v) = e(v, X)$. Note that this model generalizes the previously discussed h-models. Let e^r be the *inverse* function of e such that $e^r(e(v), v) = X$. We denote an h-model as *invertible* if such an inverse function e^r exists. Theorem 4 shows that if the h-model is invertible and e^r is monotonic, then $P_{gen}(n) = e^r(C - g(n), h(n))$ is a POF.

Theorem 4. *For any i.i.d. random variable X and invertible h-model $e(v) = e(v, X)$, if e^r is monotonic then P_{gen} is a POF.*

Proof. For any pair of nodes n_1, n_2, if $PT_{h,C}(n_1) \ge PT_{h,C}(n_2)$ then according to Corollary 1:

$$\Pr\big(e\big(h(n_1)\big) < C-g(n_1)\big) \ge \Pr\big(e\big(h(n_2)\big) < C-g(n_2)\big)$$
$$\Pr\big(e\big(h(n_1), X\big) < C-g(n_1)\big) \ge \Pr\big(e\big(h(n_2), X\big) < C-g(n_2)\big) \quad \big[e\big(h(n)\big) = e\big(h(n), X\big)\big]$$
$$\Pr\big(X < e^r\big(C-g(n_1), h(n_1)\big)\big) \ge \Pr\big(X < e^r\big(C-g(n_2), h(n_2)\big)\big) \quad [h\text{-model is invertible}]$$
$$e^r\big(C-g(n_1), h(n_1)\big) \ge e^r\big(C-g(n_2), h(n_1)\big) \quad \big[X \text{ i.i.d., } e^r \text{ monotone}\big]$$
$$P_{gen}(n_1) \ge P_{gen}(n_2)$$

The reverse direction is straightforward. □

Table 7
h models and their corresponding functions.

h-model ($e(v)$)	$e^r(e(v), v)$	$P_{gen}(n) = e^r(C - g(n), h(n))$
$v + X$ (additive)	$e(v) - v$	$C - g(n) - h(n)$
$v \cdot X$ (linear relative)	$\frac{e(v)}{v}$	$\frac{C-g(n)}{h(n)}$
v^X	$\log_v(e(v))$	$\log_{h(n)}(C - g(n))$

Notice that Lemma 1 and Theorem 3 are special cases of Theorem 4. Table 7 presents a few examples of how Theorem 4 can be used to obtain corresponding POFs for various h-models.

The exact h-model is domain and heuristic dependent. Analyzing a heuristic in a given domain and identifying its h-model may be done analytically in some domains with explicit knowledge about the domain. Another option for identifying an h-model is to add a preprocessing stage in which a set of problem instances are solved optimally, and the h-model is discovered using curve fitting methods. In our experiments, we found the linear-relative h-model to be sufficiently accurate and saw no justification for using more complex h-models. This option is given since such cases may theoretically exist.

7. Conclusions and future work

This paper discussed two search problems: bounded-cost search, and anytime search. In a bounded-cost search, once a solution of cost lower than a given cost is found, the search can halt. In an anytime search, the search continues, returning better and better solutions until the search is either halted by the user, or until an optimal solution has been found. Both types of problems have important applications and this paper presents PTS and APTS/ANA* to solve them.

We showed that an efficient bounded-cost search algorithm can be easily converted to an efficient anytime search algorithm. This is done by running the bounded-cost search algorithm iteratively, giving the bounded-cost search algorithm a lower cost bound in every iteration. An attractive property of the resulting algorithm is that there is no need for parameter tuning, in contrast to most common anytime search algorithms.

The relation between bounded-cost and anytime search problems emphasizes the need for an efficient bounded-cost search algorithm. In this paper we propose such an algorithm, called PTS. PTS is a best–first search that expands nodes in order of their potential, which is the probability that a node will lead to a solution under the cost bound. The relation between a given heuristic and the optimal cost is used to develop a cost function that can order the nodes in *OPEN* approximately according to their potential, without actually calculating it. Both PTS and its corresponding anytime algorithm APTS/ANA* outperform competitive algorithms on a wide range of domains.

7.1. BEES, BEEPS, AEES, and incorporating distance estimates

The heuristic $h(\cdot)$ estimates the minimum cost of reaching a goal. Recent work has shown that in domains with non-unit edge cost, $h(\cdot)$ may not be correlated with the number of actions needed to reach a goal [39]. In such cases, prior work has shown that it is very beneficial to incorporate into the search algorithm an additional heuristic that estimates the minimum actions needed to reaching a goal [46]. Such a "distance" heuristic, commonly denoted as $d(\cdot)$, together with an online learned inadmissible heuristic $\hat{h}(\cdot)$, has been used in a new suboptimal search algorithm called EES [20], a new anytime search algorithm called AEES [63] and a new bounded-cost search algorithm called BEES [40].

BEES was shown to outperform PTS in domains with non-unit edge costs, including "Zenotravel" and "Elevators", two domains from the International Planning Competition (IPC) that were solved with a domain independent planner. Follow up work showed that in several other IPC benchmarks with non-unit edge cost, PTS can even be outperformed by a pure heuristic search [34]. Similar results appear when comparing AEES to APTS/ANA* [63].

However, PTS is competitive with BEES in domains with unit edge costs. See the results for the Tiles domain (which is the 15-puzzle) [40] and Tinybot domain [34], which were the only unit edge-cost domains they experimented with.[17] When PTS is as good as BEES, one would probably prefer using PTS since implementing it is much simpler: BEES requires maintaining at least two open lists as well as online learning an improved heuristic, while PTS is a simple best–first search algorithm with an easy to compute evaluation function. Similar reasoning applies for comparing APTS/ANA* and AEES.

This emphasizes the need for PTS to consider $d(\cdot)$ in domains with non-unit edge cost. A first attempt to do so was in the BEEPS algorithm [63], which is equivalent to BEES except for using the PTS evaluation function when all nodes in OPEN are estimated to be above the bound. Empirically, this combination of BEES and PTS performed almost exactly like BEES. Thus, incorporating $d(\cdot)$ into PTS in an effective manner is still an open challenge. A possible direction for doing this is by considering explicitly the probabilistic relation between $h(n)$, $d(n)$ and $h^*(n)$. For example, one could apply machine learning techniques to learn this relation from past experience.

[17] Further experimental results for other domain-independent planning domains with unit edge-cost are needed to provide a conclusive statement about the performance of PTS in domain-independent planning.

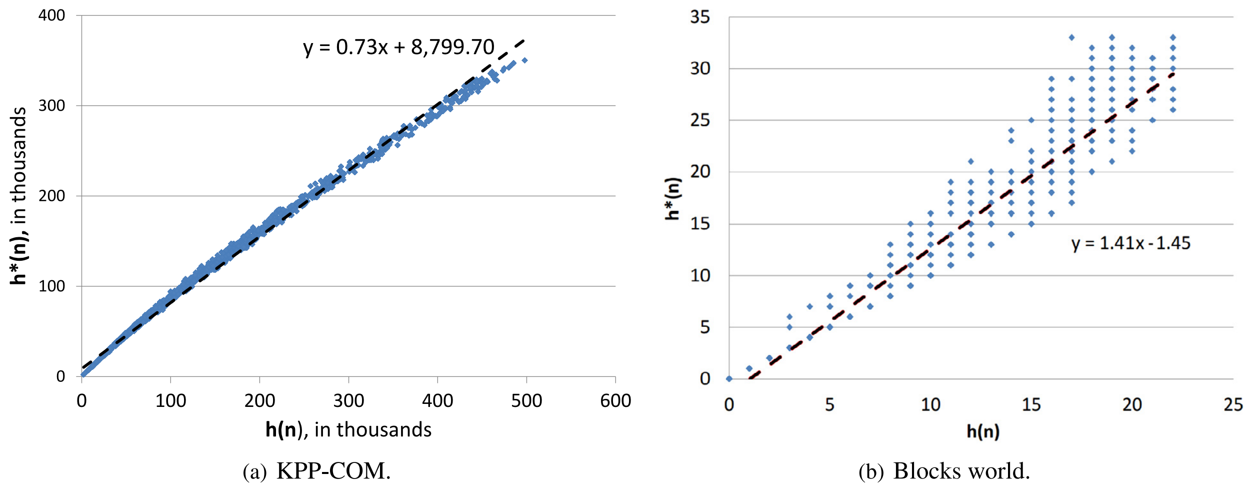


Fig. 12. Optimal solution vs. heuristic for the KPP-COM and blocksworld domains.

7.2. Expected search effort instead of potential

PTS aims at expanding the node with the highest probability to lead to a solution under the bound. The task in a bounded-cost search is, however, to find such a solution as fast as possible. Thus, we might also consider the expected effort of finding such a solution, and not just the probability that such a solution exists. Estimating search effort may be done with search effort prediction formulas and algorithms [64–66]. Future work will investigate how to incorporate such estimates, in addition to the potential of a node. For example, a node that is very close to a goal might be preferred to a node that has a slightly higher potential but is farther from a goal.

This paper combines results from two research groups who independently discovered APTS/ANA*. One group derived the anytime search algorithm APTS by studying bounded-cost search problems [7,8]. The other research group derived ANA* motivated by a desire to avoid parameter tuning [9]. Both groups had papers under review simultaneously and subsequently published in 2011. We thank Wheeler Ruml for pointing out this relationship.

Acknowledgements

This research was supported by the Israel Science Foundation (ISF) under grant number 417/13 to Ariel Felner. This work was supported in part (for Goldberg and van den Berg) by the US National Science Foundation under Award IIS-1227536. We thank Maxim Likhachev for making his implementation of ARA* available to us, as well as publishing the code for ANA*. Our implementation of ANA* is freely available in his Search-based Planning Library (SBPL) at: http://www.sbpl.net.

Appendix A. Domains and their *h*-models

In this appendix we show the *h*-models of two domains and heuristics. These models were obtained by optimally solving a set of problem instances. Then we backtracked from the goal to the start, and for every node on that path plotted its h^* value versus its heuristic value (according to the selected heuristic).

A.1. Key Player Problem in Communication (KPP-COM)

Fig. 12(a) shows the h-to-h^* relation for 100 KPP-COM problem instances. See Section 5.1.2 for details on this domain and the heuristic we used for it. Each problem instance is a graph, randomly generated according to the Barabási–Albert model [54] (the choice of this type of graphs is motivated in Section 5.1), having 700 nodes and a density factor of 3.

The dashed black line in Fig. 12(a) is a linear fit of the data. As can be seen, a slight curve (e.g., a logarithmic model) would fit nicely. However, the linear fit is simpler and is also quite accurate.

A.2. Blocks world

Next, we analyzed the h-to-h^* relation for the blocks world domain, which is one of the well-known planning domains. We used the Fast Downward planner [67] with the admissible LM-cut heuristic [68]. Fig. 12(b) shows the h-to-h^* relation for all the blocks world problem instances from the International Planning Competition '06, which are publicly available in the Fast Downward software suite. Again, the dashed black line is a linear fit of the data. As can be seen, this heuristic in this domain also exhibits a clear *linear relative h-model.*

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. SSC-4 (2) (1968) 100–107.
[2] R.E. Korf, Depth-first iterative-deepening: an optimal admissible tree search, Artif. Intell. 27 (1) (1985) 97–109.
[3] M. Helmert, G. Röger, How good is almost perfect? in: AAAI, 2008, pp. 944–949.
[4] S. Zilberstein, Using anytime algorithms in intelligent systems, AI Mag. 17 (3) (1996) 73–83.
[5] R.A. Valenzano, N.R. Sturtevant, J. Schaeffer, K. Buro, A. Kishimoto, Simultaneously searching with multiple settings: an alternative to parameter tuning for suboptimal single-agent search algorithms, in: ICAPS, 2010, pp. 177–184.
[6] M. Likhachev, G.J. Gordon, S. Thrun, ARA*: anytime A* with provable bounds on sub-optimality, in: NIPS, 2003.
[7] R. Stern, R. Puzis, A. Felner, Potential search: a new greedy anytime heuristic search, in: SoCS, 2010, pp. 119–120.
[8] R. Stern, R. Puzis, A. Felner, Potential search: a bounded-cost search algorithm, in: ICAPS, 2011, pp. 234–241.
[9] J. van den Berg, R. Shah, A. Huang, K.Y. Goldberg, Anytime nonparametric A*, in: AAAI, 2011, pp. 105–111.
[10] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 3rd edition, Prentice-Hall, Englewood Cliffs, NJ, 2010.
[11] R. Stern, T. Kulberis, A. Felner, R. Holte, Using lookaheads with optimal best–first search, in: AAAI, 2010, pp. 185–190.
[12] E.W. Dijkstra, A note on two problems in connexion with graphs, Numer. Math. 1 (1959) 269–271.
[13] A. Felner, Position paper: Dijkstra's algorithm versus uniform cost search or a case against Dijkstra's algorithm, in: SOCS, 2011, pp. 47–51.
[14] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 2nd edition, The MIT Press, 2001.
[15] R.E. Korf, Linear-space best–first search, Artif. Intell. 62 (1) (1993) 41–78.
[16] I. Pohl, Heuristic search viewed as path finding in a graph, Artif. Intell. 1 (3–4) (1970) 193–204.
[17] I. Pohl, The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving, in: IJCAI, 1973, pp. 12–17.
[18] J. Pearl, J. Kim, Studies in semi-admissible heuristics, IEEE Trans. Pattern Anal. Mach. Intell. 4 (4) (1982) 392–400.
[19] J.T. Thayer, W. Ruml, Faster than weighted A*: an optimistic approach to bounded suboptimal search, in: ICAPS, 2008, pp. 355–362.
[20] J.T. Thayer, W. Ruml, Bounded suboptimal search: a direct approach using inadmissible estimates, in: IJCAI, 2011, pp. 674–679.
[21] D. Furcy, S. Koenig, Limited discrepancy beam search, in: IJCAI, 2005, pp. 125–131.
[22] E.A. Hansen, R. Zhou, Anytime heuristic search, J. Artif. Intell. Res. 28 (2007) 267–297.
[23] E. Balas, P. Toth, Branch and bound methods, in: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shwoys (Eds.), Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization, Wiley, Chichester, 1985.
[24] S. Richter, J.T. Thayer, W. Ruml, The joy of forgetting: faster anytime search via restarting, in: ICAPS, 2010, pp. 137–144.
[25] S. Aine, P.P. Chakrabarti, R. Kumar, AWA* – a window constrained anytime heuristic search algorithm, in: IJCAI, 2007, pp. 2250–2255.
[26] R. Zhou, E.A. Hansen, Beam-stack search: integrating backtracking with beam search, in: ICAPS, 2005, pp. 90–98.
[27] J.T. Thayer, W. Ruml, Anytime heuristic search: frameworks and algorithms, in: SOCS, 2010, pp. 121–128.
[28] R. Taig, R.I. Brafman, Compiling conformant probabilistic planning problems into classical planning, in: ICAPS, 2013, pp. 197–205.
[29] A. Stern, I. Dagan, A confidence model for syntactically-motivated entailment proofs, in: RANLP, 2011, pp. 455–462.
[30] A. Stern, R. Stern, I. Dagan, A. Felner, Efficient search for transformation-based inference, in: ACL, 2012, pp. 283–291.
[31] A. Zanarini, G. Pesant, Solution counting algorithms for constraint-centered search heuristics, Constraints 14 (2009) 392–413.
[32] P. Haslum, H. Geffner, Heuristic planning with time and resources, in: European Conference on Planning (ECP), vol. 1, 2001, pp. 121–132.
[33] H. Nakhost, J. Hoffmann, M. Müller, Improving local search for resource-constrained planning, in: SOCS, 2010, pp. 81–82.
[34] P. Haslum, Heuristics for bounded-cost search, in: ICAPS, 2013, pp. 312–316.
[35] J. Pearl, Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley Pub. Co., Inc., Reading, MA, 1984.
[36] A. Felner, R.E. Korf, S. Hanan, Additive pattern database heuristics, J. Artif. Intell. Res. 22 (2004) 279–318.
[37] U. Sarkar, P. Chakrabarti, S. Ghose, S. De Sarkar, Reducing reexpansions in iterative-deepening search by controlling cutoff bounds, Artif. Intell. 50 (2) (1991) 207–221.
[38] E. Burns, W. Ruml, Iterative-deepening search with on-line tree size prediction, in: LION, 2012, pp. 1–15.
[39] C.M. Wilt, W. Ruml, When does weighted A* fail? in: SOCS, 2012, pp. 137–144.
[40] J.T. Thayer, R. Stern, W. Ruml, A. Felner, Faster bounded-cost search using inadmissible estimates, in: ICAPS, 2012, pp. 270–278.
[41] M. Boullé, A parameter-free classification method for large scale learning, J. Mach. Learn. Res. 10 (2009) 1367–1385.
[42] G.R. Harik, F.G. Lobo, A parameter-less genetic algorithm, in: GECCO, vol. 99, 1999, pp. 258–267.
[43] A. Foss, O.R. Zaïane, A parameterless method for efficiently discovering clusters of arbitrary shape in large datasets, in: International Conference on Data Mining (ICDM), IEEE, 2002, pp. 179–186.
[44] A.J. Dionne, J.T. Thayer, W. Ruml, Deadline-aware search using on-line measures of behavior, in: SOCS, 2011, pp. 39–46.
[45] W. Ruml, M.B. Do, Best-first utility-guided search, in: IJCAI, 2007, pp. 2378–2384.
[46] J.T. Thayer, W. Ruml, Using distance estimates in heuristic search, in: ICAPS, 2009, pp. 382–385.
[47] W.E. Story, Notes on the "15" puzzle, Am. J. Math. 2 (4) (1879) 397–404.
[48] R.C. Holte, A. Felner, J. Newton, R. Meshulam, D. Furcy, Maximizing over multiple pattern databases speeds up heuristic search, Artif. Intell. 170 (16–17) (2006) 1123–1136.
[49] U. Zahavi, A. Felner, R.C. Holte, J. Schaeffer, Duality in permutation state spaces and the dual search algorithm, Artif. Intell. 172 (4–5) (2008) 514–540.
[50] M.G. Everett, S.P. Borgatti, The centrality of groups and classes, J. Math. Sociol. 23 (3) (1999) 181–201.
[51] L.C. Freeman, A set of measures of centrality based on betweenness, Sociometry 40 (1) (1977) 35–41.
[52] R. Puzis, Y. Elovici, S. Dolev, Finding the most prominent group in complex networks, AI Commun. 20 (4) (2007) 287–296.
[53] S. Dolev, Y. Elovici, R. Puzis, P. Zilberman, Incremental deployment of network monitors based on group betweenness centrality, Inf. Process. Lett. 109 (20) (2009) 1172–1176.
[54] A.L. Barabási, R. Albert, Emergence of scaling in random networks, Science 286 (5439) (1999) 509–512.
[55] D.J. Lipman, S.F. Altschul, J.D. Kececioglu, A tool for multiple sequence alignment, Proc. Natl. Acad. Sci. USA 86 (12) (1989) 4412–4415.
[56] T. Ikeda, H. Imai, Enhanced A* algorithms for multiple alignments: optimal alignments for several sequences and k-opt approximate alignments for large cases, Theor. Comput. Sci. 210 (2) (1999) 341–374.
[57] A.S. Konagurthu, P.J. Stuckey, Optimal sum-of-pairs multiple sequence alignment using incremental Carrillo and Lipman bounds, J. Comput. Biol. 13 (3) (2006) 668–685.
[58] T. Yoshizumi, T. Miura, T. Ishida, A* with partial expansion for large branching factor problems, in: AAAI, 2000, pp. 923–929.
[59] T. Ikeda, H. Imai, Fast A* algorithms for multiple sequence alignment, in: Workshop on Genome Informatics, 1994, pp. 90–99.
[60] H. Kobayashi, H. Imai, Improvement of the A* algorithm for multiple sequence alignment, in: Genome Informatics Series, 1998, pp. 120–130.
[61] W. Zhang, R.E. Korf, Performance of linear-space search algorithms, Artif. Intell. 79 (2) (1995) 241–292.
[62] W. Zhang, Depth-first branch-and-bound versus local search: a case study, in: AAAI/IAAI, 2000, pp. 930–935.

[63] J.T. Thayer, J. Benton, M. Helmert, Better parameter-free anytime search by minimizing time between solutions, in: SOCS, 2012, pp. 120–128.
[64] L.H.S. Lelis, S. Zilles, R.C. Holte, Predicting the size of IDA*'s search tree, Artif. Intell. 196 (2013) 53–76.
[65] L. Lelis, R. Stern, A. Felner, S. Zilles, R.C. Holte, Predicting optimal solution cost with bidirectional stratified sampling, in: ICAPS, 2012, pp. 155–163.
[66] R.E. Korf, M. Reid, S. Edelkamp, Time complexity of iterative-deepening-A*, Artif. Intell. 129 (1–2) (2001) 199–218.
[67] M. Helmert, The fast downward planning system, J. Artif. Intell. Res. 26 (2006) 191–246.
[68] M. Helmert, C. Domshlak, Landmarks, critical paths and abstractions: what's the difference anyway? in: ICAPS, 2009, pp. 162–169.