Faster Bounded-Cost Search Using Inadmissible Estimates

Jordan T. Thayer
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
jtd7 at *cs.unh.edu*

Roni Stern and **Ariel Felner**
Information Systems Engineering
Ben Gurion University
Beer-Sheva, Israel 85104
roni.stern at *gmail.com*, *felner* at *bgu.ac.il*

Wheeler Ruml
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
ruml at *cs.unh.edu*

Abstract

Many important problems are too difficult to solve optimally. A traditional approach to such problems is bounded suboptimal search, which guarantees solution costs within a user-specified factor of optimal. Recently, a complementary approach has been proposed: bounded-cost search, where solution cost is required to be below a user-specified absolute bound. In this paper, we show how bounded-cost search can incorporate inadmissible estimates of solution cost and solution length. This information has previously been shown to improve bounded suboptimal search and, in an empirical evaluation over five benchmark domains, we find that our new algorithms surpass the state-of-the-art in bounded-cost search as well, particularly for domains where action costs differ.

Introduction

If time and memory permit, we can use algorithms like A* (Pohl 1970) or IDA* (Korf 1985) to solve classical planning problems optimally. In many practical settings, finding optimal solutions is impractically expensive. So we turn to suboptimal solving techniques, which can quickly return solutions whose cost may be greater than optimal.

There are a number of possible settings for suboptimal search. Perhaps the best known is bounded suboptimal search: given a user-supplied suboptimality bound w, find a solution guaranteed to be no more than w times more expensive than optimal as quickly as possible. The most famous algorithm for this setting is weighted A* (Pohl 1970). Relaxing the suboptimality bound w reduces the difficulty of proving that a solution is cheap enough, usually reducing the time required to solve the problem.

Recently, Stern, Puzis, and Felner (2011) introduced a new setting for suboptimal search called *bounded-cost search*: given a user-specified cost bound C, find a plan with cost less than or equal to C as fast as possible. Bounded-cost search corresponds to many realistic cost-constrained scenarios such as planning under a fixed budget for plan execution. Stern, Puzis, and Felner also introduced an algorithm called Potential search (PTS) specifically designed for the bounded-cost search. PTS is a best-first search on potential — the probability that a given node will be part of a solution whose cost is within the user-supplied budget C. Nodes more likely to have a goal node beneath them are preferred.

In this paper, we note that PTS has a shortcoming. While it considers how likely a node is to have a goal beneath it, it ignores the effort of actually finding that goal. If we had both of these estimates, we could optimize the goal of bounded-cost search directly: find a solution within the cost bound C as quickly as possible.

After presenting PTS in more detail, we show that it can use inadmissible heuristics to improve search performance. We call this variant $\widehat{PTS}$. We then introduce a new search algorithm, Bounded-Cost Explicit Estimation Search (BEES), that considers an estimate of the search effort beneath a node. It does this by using a heuristic estimate of the number of actions between a node and a goal, denoted by $\widehat{d}$, which is used as a proxy for remaining search effort. We also introduce a variant of BEES that considers both the potential of a node and $\widehat{d}$. This is called Bounded-Cost Explicit Estimation Potential Search (BEEPS). Experimental results in five benchmark domains show that BEES and BEEPS perform nearly identically to the previous state-of-the-art PTS algorithm when actions have unit cost and outperform the previous state-of-the-art by up to four orders of magnitude in domains where actions have differing costs. We also show that algorithms designed for the bounded-cost setting outperform modified bounded suboptimal search algorithms as well as simple baselines in the bounded-cost setting.

Taken together with the previous work on bounded suboptimal search, these results on bounded-cost search suggest that inadmissible estimates of solution cost and solution length are widely useful for fast suboptimal heuristic search.

The Bounded-Cost Search Problem

Stern, Puzis, and Felner (2011) define bounded-cost search in the context of heuristic shortest-path graph search: Given a description of a state space, a start state s, a goal test function and a constant C, find a path from s to a goal state with cost less than or equal to C.

One might be tempted to view a bounded-cost search problem as a constraint satisfaction problem (CSP), where the desired cost bound is simply a constraint on the solution cost. However, we address the more general problem of finding a path in a search space, where the search space

can be defined implicitly by an initial state and an expand, or successor, function. Such an implicitly defined search space can be infinitely large. This prohibits solving bounded-cost problems with CSP solvers, as they require that all decision variables be defined up front. The algorithms we develop in this paper belong to the best-first search framework, and are suited to search in potentially infinite search spaces.

Bounded-cost search problems is related to resource-constraint planning (Haslum and Geffner 2001; Nakhost, Hoffmann, and Müller 2010). In resource-constraint planning there are limited resources, and actions may consume these resources. The task to find a feasible plan (with respect to the resources) with minimal make-span. This is similar to bounded-cost search, in which there is only type resources which is the cost of the plan. In contrast, in bounded-cost search we do not want to minimize the plan make-span - any plan under the cost bound is good enough. Thus, bounded-cost algorithms are expected to outperform the more general resource-constraint planner. Other related work limited the applicable operators as a function of the cost (Phillips and Likhachev 2011). This type of planning can be reduced to a bounded-cost search problem, by adding a single goal-achieving operator that can only be applied if the cost is below the bound. However, in this setting too the task is to optimize the plan length/cost, while in bounded-cost search any solution under the bound is acceptable.

Potential Search

To our knowledge, PTS is the first best-first heuristic search algorithm to address the bounded-cost search problem. PTS is based on considering the potential of all nodes that have been generated but not yet expanded (i.e. nodes on OPEN). The potential of a node is the probability that a solution of cost no more than C exists beneath that node. Let $h^*(n)$ be the true cost of a path from n to a goal. Then the *potential* of node n is $PT_C(n) = Pr(g(n) + h^*(n) \leq C)$.

PTS is simply a best-first search on PT_C. Explicitly calculating the potential of a node is challenging. However, for some cases it is possible to order the nodes according to their potential without calculating it exactly. This can be done if one knows the relation between the real cost-to-go $h^*(n)$ and its lower-bound $h(n)$. This relation between h and h^* is called the heuristic error model. Several heuristic functions have shown to exhibit an approximately *linear-relative heuristic error* model, which means $h^*(n)$ has a distribution similar to $h(n)$ times an i.i.d random variable α.

Stern, Puzis, and Felner (2011) proved that, for the case of the linear-relative heuristic error, the node with the highest potential is the node with minimum $f_{lnr}(n) = \frac{h(n)}{C-g(n)}$. In this paper we assume the linear relative heuristic error model holds, and therefore PTS is a best-first search using f_{lnr}.

Potential Search with Inadmissible Estimates

While not stated explicitly, PTS was always described in the context of admissible heuristics. Admissible heuristics are lower bounds on the cost-to-go. Inadmissible heuristics may return estimates that are larger than the true cost-to-go, but they tend to be more accurate than admissible heuristics. Inadmissible heuristics can be hand-crafted, learned offline from training data (Samadi, Felner, and Schaeffer 2008), or learned during search (Thayer, Dionne, and Ruml 2011).

The first contribution of this paper is to point out that indeed inadmissible heuristics can be used in bounded-cost search. Let $\hat{h}(n)$ denote an inadmissible heuristic. If an inadmissible heuristic function $\hat{h}(n)$ has a linear-relative error model then PTS can be implemented using $\hat{f}_{lnr}(n) = \frac{\hat{h}(n)}{C-g(n)}$. $\widehat{PTS}$ denotes PTS with inadmissible heuristics.

In practice, we implement $\widehat{PTS}$ as a best-first search on $\hat{f}_{lnr}(n) = \frac{\hat{h}(n)}{1-\frac{g(n)}{C}}$, where we have effectively divided the potential score of all nodes by the cost bound C. This does not affect node ordering. Restating the node ordering function this way does two things. First, it makes it clear that for large values of C, PTS and $\widehat{PTS}$ will behave like a greedy search on cost-to-go estimates (sometimes called pure heuristic search). Secondly, it avoids precision issues caused by large C values. For large C, implementing $f_{lnr}(n)$ as $\frac{h(n)}{C-g(n)}$ will result in $f_{lnr}(n) = 0$ for all nodes. For $f_{lnr}(n) = \frac{h(n)}{1-\frac{g(n)}{C}}$, we have $f_{lnr}(n) = h(n)$.

If an admissible $h(n)$ is available, $\widehat{PTS}$ can use it for pruning. Every node generated by $\widehat{PTS}$ such that $g(n) + h(n) > C$ can immediately be pruned, as it can not lead to a solution within the desired bound. $\widehat{PTS}$ is more informed than PTS, as it uses both the admissible and the inadmissible heuristics when it searches.

Limitations of the Potential-Based Approach

Note that PTS and $\widehat{PTS}$ only consider the probability that a node is on a path to a goal whose cost is within the cost bound C. Search algorithms for the bounded-cost problem can also consider additional information, for example $\widehat{d}(n)$, an estimate of the number of actions between a node and the goal, when choosing which node to expand next. To see how this might be helpful, consider the graph depicted in Figure 1. Assume that nodes A and B are in the open list and the search algorithm needs to choose which node to expand first. While node B has a higher potential ($PT_C(B) = 0.9 > PT_C(A) = 0.89$), A is estimated to be much closer to the goal than B ($d^*(A) = 1 < d^*(B) = 10$). Clearly, PTS will expand B first, since it only considers the potential of a node. Since the goal under A is estimated to be much closer than the goal under B, expanding A is intuitively a better option. Its potential is close to the potential of B, but it is expected to return a solution much faster since it is closer to a goal. This example demonstrates an oversight in the approach taken by PTS. The task in a bounded-cost problem is to find a path to a goal under the bound as fast as possible, i.e. with minimal search effort. However, PTS considers only the potential of a node, i.e. the probability of a node being part of path to a goal under the bound. It does not account for the time needed to find that solution.

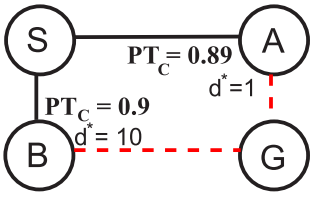

Figure 1

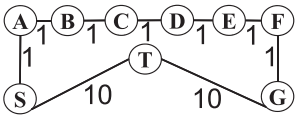

Figure 2

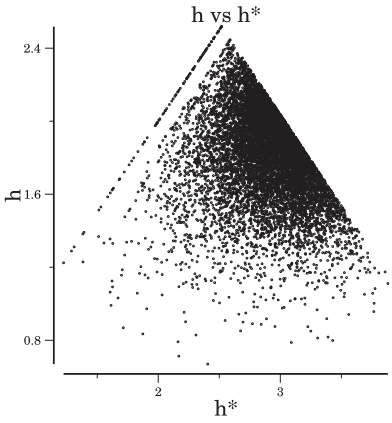

Figure 3

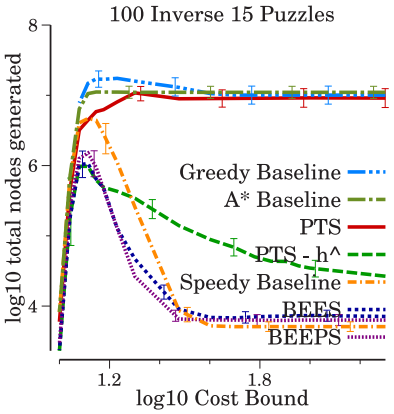

Figure 4

Bounded-Cost Explicit Estimation Search

In best-first search, the running time of the search algorithm is usually proportional to the number of nodes expanded until the goal is found. It is difficult to directly estimate the search effort that will be required to find a goal under a given node. However, in many domains it is possible to obtain an estimate of the number of actions between a node and a goal. Lower bounds of actions-to-go will be denoted by d, $\widehat{d}$ denotes estimates of actions-to-go that are not necessarily lower bounds (i.e. inadmissible estimates).

Actions-to-go is no harder to estimate than cost-to go. Heuristics are typically computed by solving some relaxed problem. $h(n)$ is the sum of the costs of the actions in the solution to this relaxed problem. When action costs differ, we can construct $d(n)$ by counting the number of actions in the solution to this relaxed problem.

The Explicit Estimation Search (EES) algorithm (Thayer and Ruml 2011), proposed for the bounded-suboptimal search setting, uses $\widehat{d}$ to solve problems quickly. The BEES and BEEPS algorithms presented in this paper are based on EES, and therefore we first discuss EES in detail.

Explicit Estimation Search

EES is designed for bounded suboptimal search in which the goal is to find a solution whose cost is within a user-specified factor of optimal as quickly as possible. EES attempts to estimate which nodes are likely to lead to solutions within the bound. Of these, it prefers to expand next that node which appears to be nearest to completion. EES must also take care to only expand nodes that can be shown to lead to a solution within the suboptimality bound, typically denoted w.

In addition to the standard admissible cost-to-go heuristic h, EES makes use of an inadmissible estimate of cost-to-go, $\widehat{h}$, as well as $\widehat{d}$, the previously discussed actions-to-go estimate. EES uses its heuristics to form a lower bound on the total cost of a solution through a node, $f(n) = g(n) + h(n)$ as well as an unbiased inadmissible estimate of the cost of a solution through a node, $\widehat{f}(n) = g(n) + \widehat{h}(n)$.

Inadmissible heuristics have many potential sources (Samadi, Felner, and Schaeffer 2008; Jabbari Arfaee, Zilles, and Holte 2011; Thayer, Dionne, and Ruml 2011). We use the well-known FF heuristic (Hoffmann and Nebel 2001) for $\widehat{h}$ and $\widehat{d}$ in domain independently planning, and we use *online corrections* proposed by Thayer, Dionne, and Ruml (2011) to construct inadmissible heuristics in our other benchmarks. These online corrections work by estimating the average error in heuristics in a single expansion. We simply need to correct for the measured error for every step we intend to take, estimated by $\widehat{d}$.

Using the heuristic functions just described, EES keeps track of three special nodes: $best_f$, which provides a lower bound on the cost of an optimal solution, $best_{\widehat{f}}$, which provides an estimate of the cost of an optimal solution to the problem, and $best_{\widehat{d}}$, the node estimated to be within the suboptimality bound w and have the smallest number of actions remaining to reach a goal. Formally:

$$best_f = \operatorname*{argmin}_{n \in open} f(n)$$

$$best_{\widehat{f}} = \operatorname*{argmin}_{n \in open} \widehat{f}(n)$$

$$best_{\widehat{d}} = \operatorname*{argmin}_{n \in open \wedge \widehat{f}(n) \leq w \cdot \widehat{f}(best_{\widehat{f}})} \widehat{d}(n)$$

At every expansion, EES chooses from among these three nodes using the following rule:

1. **if** $\widehat{f}(best_{\widehat{d}}) \leq w \cdot f(best_f)$ **then** $best_{\widehat{d}}$
2. **else if** $\widehat{f}(best_{\widehat{f}}) \leq w \cdot f(best_f)$ **then** $best_{\widehat{f}}$
3. **else** $best_f$

The intuition behind EES is that it attempts to pursue the nearest solution that is estimated to be within the suboptimality bound, that is it considers $best_{\widehat{d}}$ first. If it cannot pursue this node, it considers either improving the quality of nodes from the focal list (expanding $best_{\widehat{f}}$) or raising the lower bound (expanding $best_f$). The former addresses a flaw in A^*_ϵ (Thayer, Ruml, and Kreis 2009) and the latter ensures that we can prove that solutions returned by EES obey the suboptimality bound.

EES has been shown to find solutions faster than other bounded suboptimal search algorithms for a given suboptimality bound, especially in domains with actions of differ-

ing costs. However, EES is a bounded-suboptimal search algorithm. Thus, EES cannot be used to solve bounded-cost search problems. Next we show how we can apply the ideas that were shown to be effective in EES to construct two new efficient bounded-cost search algorithms.

Bounded-Cost Explicit Estimation Search

Bounded-Cost Explicit Estimation Search (BEES) considers both admissible and inadmissible estimates of cost-to-go (h and $\widehat{h}$) as well as inadmissible estimates of actions-to-go ($\widehat{d}$). BEES is inspired by EES in that both rely on estimates of solution cost and actions remaining to guide search rather than exclusively relying on lower bounds as PTS does. To suit the goal of bounded-cost search, instead of considering $best_{\widehat{d}}$ like EES, BEES considers the following node:

$$best_{\widehat{d}_C} = \operatorname*{argmin}_{n \in open \wedge \widehat{f}(n) \leq C} \widehat{d}(n)$$

Note that $best_{\widehat{d}_C}$ is a member of the set of all nodes in OPEN whose estimated total cost is less than that of the cost bound. Of these, $best_{\widehat{d}_C}$ is the node with the smallest $\widehat{d}(n)$. $best_{\widehat{d}_C}$ is the node we estimate has the fewest actions remaining between it and a goal, among all the nodes whose estimated total cost is less than the cost bound.

BEES chooses which node to expand according to the rule:

1. **if** there exists $n \in open$ s.t. $\widehat{f}(n) \leq C$
2. **then** return $best_{\widehat{d}_C}$
3. **else** return $best_f$

BEES chooses to expand either $best_{\widehat{d}_C}$ or $best_f$ according the rule described above. Using this rule, BEES attempts to pursue the shortest solution estimated to be within cost bound C if it estimates that such a node exists (line 2). If BEES thinks there are no solutions within the cost bound, it expands nodes in A^* order to efficiently prove no solution exists (line 3).

A Potentially Improved Rule

While straightforward, the previous approach ignores the potential measurement suggested by Stern, Puzis, and Felner, 2011. To takes this new quantity into account, we propose Bounded-Cost Explicit Estimation Potential Search (BEEPS). In addition to $best_{\widehat{d}_C}$, BEEPS considers expanding the node with the highest potential, ie lowest $\widehat{f}_{lnr}(n)$.

The node selection strategy of BEEPS is exactly the same as that of BEES, differing only in the last line. When BEES decides to return $best_f$, BEEPS will return $best_p$. BEES assumes that $\widehat{f}$ is accurate and so if $best_{\widehat{d}_C}$ does not exist, then there must not exist a solution to this problem within cost bound C. If that is true, then the optimal way of proving it is by expanding nodes in A* order until we have shown that there is no node with $f(n) \leq C$, proving the problem unsolvable. In contrast, BEEPS acknowledges that $\widehat{h}$ is not a perfect estimator and thus even when $best_{\widehat{d}_C}$ does not exist, the problem may well be solvable. For solvable bounded-cost problems, PTS was shown to be superior to A* with pruning (Stern, Puzis, and Felner 2011), so BEEPS reverts to $\widehat{PTS}$ instead.

Comparison of Algorithms

Next, we note several observations on the comparative behavior of BEES, BEEPS, PTS and $\widehat{PTS}$.

Unsolvable Instances For a given problem and cost bound, there are two possibilities, either there exists a solution with cost no more than C, or no such solution exists. First, consider the case in which there is no solution under the cost bound. Proving that no solution exists requires expanding all the nodes with $f(n) \leq C$. Thus, every algorithm will have to expand all the nodes with $f(n) \leq C$, and there is no need to expand any node with $f(n) > C$.

Consequently, every bounded-cost search algorithm we consider prunes nodes with $f(n) > C$. Thus, when no solution below the desired cost bound exists, BEES, BEEPS, $\widehat{PTS}$ and PTS will all expand exactly the same set of nodes: those with $f(n) \leq C$.

However, in graphs with transpositions or cycles, where there are multiple paths from the root to the same state, nodes may be expanded more than once. This may occur when a node is reached by a suboptimal path (i.e., with $g(n)$ larger than the lowest-cost path from the root to n). That node may be expanded many times if it is expanded before the lowest-cost path to it is found.

Observation 1 *If there is no solution with cost smaller than or equal to the cost bound C, the relative performance of the algorithms will relate to the number of nodes they re-expand. If the number of re-expansions is small with respect to the number of nodes with $g(n) + h(n) \leq C$, we expect BEES, BEEPS, $\widehat{PTS}$ and PTS to perform similarly.*

The amount of these node re-expansions depend on the structure of the domain (e.g., number of cycles), as well as the type of the search algorithm. For example, with an admissible and consistent heuristic, A^* is guaranteed to never re-expand a node (Pearl 1984), while greedy search may re-expand a node many times.

When no solution exists, if $h = h^*$ then all algorithms will end immediately saying that there is no solution of cost less than C. Assuming that $h = h^*$ is unrealistic, but it is common that the inadmissible cost-to-go estimate $\widehat{h}$ is more accurate than h. Thus, it is interesting to consider another theoretical scenario, where $h \neq h^*$ but $\widehat{h} = h^*$.

In this setting, both BEES and BEEPS will use only their second expansion rule as all the nodes will have $\widehat{f}(n) > C$ (since there is no solution with cost C). Consequently, BEES will perform A* (always expanding $best_f$) and thus do the least amount of work possible to show that no solution exists. On the other hand, BEEPS and $\widehat{PTS}$ will expand nodes in $\hat{f}_{lnr}(n)$ order, which is not identical to the A* order and is thus inefficient for proving no solution exists. Similarly, PTS will also be inefficient for proving no solution exists, as it expands nodes according to f_{lnr}. So in this very optimistic setting, BEES will be the most effective algorithm. Realistically, $\widehat{h} \neq h^*$ and therefore BEES

may expand some nodes in a different order than A*, even if there is no solution. This may lead to suboptimal performance. If $\widehat{h}$ is fairly accurate and there are many possible paths to reach a node, then BEES will expand in the same order as A* for most of the nodes, and will therefore outperform BEEPS when there is no solution under the cost bound.

Solvable Instances: Non-Uniform Action Costs Next, we discuss the case where there is a solution of cost smaller than or equal to the given cost bound C. First, consider the more general case where there are non-uniform edge costs.

Figure 2 provides a concrete example of a problem where both BEES and BEEPS will outperform the previous state-of-the-art, PTS. There are two paths through this graph from the start S to the goal G. There is a path with two arcs of cost 20 and a path with 7 arcs of cost 7. Assume that prefect information is available, that is $h(n) = \widehat{h}(n) = h^*(n)$ and $\widehat{d}(n) = d^*(n)$. Clearly, if the cost bound is 20 or more, the fastest bounded-cost search algorithm would expand only node S and T. However, even with such perfect information, PTS and $\widehat{PTS}$ will always find the longer solution (i.e., that which passes via 7 arcs), regardless of the cost bound. To see this, first note that because $h(n) = \widehat{h}(n)$, PTS and $\widehat{PTS}$ behave exactly the same, expanding in every iteration the node with the lowest $f_{lnr} = \frac{h^*}{C-g}$. As a result of the perfect information, f_{lnr} will decrease along any optimal path to the goal (of cost less than or equal to C). That is, $f_{lnr}(A) \leq f_{lnr}(B) \leq ... \leq f_{lnr}(F)$. Thus, if $f_{lnr}(A) \leq f_{lnr}(T)$, PTS will expand the nodes in the longer path. So long as C is at least 20, initially the open list of these algorithms will contain A and T. $f_{lnr}(A) = \frac{6}{C-1}$, while $f_{lnr}(T) = \frac{10}{C-10}$. By algebra, for every value of $C \geq 20$, $f_{lnr}(T) < f_{lnr}(A)$, and thus PTS would prefer node T to node A, and PTS finds the longer path.

For $C \geq 20$, BEES and BEEPS will find the shorter solution. $\widehat{d}$ tells these two algorithms to prefer the shorter path through T explicitly. For problems like this one, BEES and BEEPS will return solutions faster than previous approaches because they rely on information that the other algorithms do not use: estimates of remaining solution length. For this reason, in non-unit cost domains we expect the following:

Observation 2 *In domains with non-uniform edge cost, for solvable problem instances (i.e., when there is a solution below the cost bound), we expect BEES and BEEPS to find solutions faster than PTS and $\widehat{PTS}$ for the same cost bound.*

Solvable Instances: Uniform Action Costs Next, we consider the case where there exists a solution of cost smaller than or equal to C, and the domain has only uniform edge costs. In this case, cost-to-go estimates and actions-to-go estimates are the same (i.e, $d(n) = h(n)$ and $\widehat{d}(n) = \widehat{h}(n)$). Consequently, greedy search on h is equivalent to greedy search on d (and similarly for $\widehat{h}$ and $\widehat{d}$).

Generally speaking, for large enough values of C, both BEES and BEEPS will always apply the first expansion rule which means that they will follow the heuristic greedily to a goal. Similarly, for large values of C, ordering nodes according to the cost functions used by $\widehat{PTS}$ and PTS converge to the same ordering as greedy search (i.e., on $\widehat{h}$ for $\widehat{PTS}$ and on h for PTS). We have perfect information (i.e., $h = \widehat{h} = h^*$), BEES and BEEPS converge to greedy search on $\widehat{h}$ and will expand exactly those nodes that lie on an optimal solution. Since $\widehat{h} = h^*$, this is optimal. Similarly, PTS converges to the same greedy search. This leads to the following observations.

Observation 3 *For values of C that are significantly larger than the optimal solution, in domains with uniform cost edges, where either the admissible heuristic is accurate or where solutions under C are easy to find, BEES, BEEPS, $\widehat{PTS}$ and PTS should expand a similar number of nodes.*

Overhead The discussion above has been in terms of nodes evaluated, but BEES, BEEPS and $\widehat{PTS}$ use more heuristics and node orderings (e.g. heap data structures) than PTS. Specifically, PTS only considers the admissible heuristic h, while $\widehat{PTS}$ considers the inadmissible heuristic $\widehat{h}$, and BEES and BEEPS consider $\widehat{d}$ as well.

Observation 4 *The overhead per node for PTS is expected to be lower than that for $\widehat{PTS}$, which is expected to be lower than that of either BEES or BEEPS.*

In summary, we expect that in domains with uniform edge cost, BEES, BEEPS, $\widehat{PTS}$ and PTS will have similar performance in the cases described in Observations 1 and 3, while for non-uniform edge costs, we expect BEES and BEEPS to outperform PTS and $\widehat{PTS}$ for solvable instances (Observation 2). By Observation 4, we should expect that, for domains with uniform cost actions, the reduced overhead of PTS would lead to slightly better performance in terms of time to find a solution.

Empirical Evaluation

In order to evaluate whether these theoretical differences in search performance manifest in practice, we performed an empirical evaluation of PTS, $\widehat{PTS}$, BEES, and BEEPS. All experiments were run on 64-bit Intel Linux systems with 3.16 GHz Core2 duo processors and 8 GB of RAM. The domain specific solvers, discussed first, were implemented in OCaml. The evaluation in domain independent planning was done in the FastDownward framework (Helmert 2011) and algorithms are implemented in C++. Algorithms were cut off when they exhausted memory or spent more than ten minutes in the domain-specific solvers or half an hour in FastDownward on a given instance.

Vacuum World This domain mirrors the first state space presented in Russell and Norvig (2010, page 36). A robot is charged with cleaning up a grid world. Movement is in the four cardinal direction and whenever the robot is on top of a pile of dirt, it may vacuum. The dirt has weight, making it harder for the robot to move: the cost for the robot to take any action (including vacuuming) is one plus the number of dirt piles the robot has cleaned up. The problem is solved when no dirt remains.

We use 150 solvable instances that are 200×200 and each cell has a 35% chance of being blocked. We place ten piles of dirt and the robot randomly in unblocked cells. For h we compute the minimum spanning tree of the robot and the dirt piles, order the edges by greatest length first, and then multiply the edge weights by the current weight of the robot plus the number of edges already considered. d is computed as a greedy traversal of the dirt piles (i.e. vacuum to nearest dirt pile, then next nearest, and so on) assuming that the room contains no obstacles. In this domain, $\widehat{h}$ and $\widehat{d}$ are computed using the online corrections discussed previously.

Figure 5 shows the performance of the bounded-cost algorithms on this search domain. In all performance plots, the x-axis shows the cost bound C. The y-axis shows either CPU time or the number of nodes expanded, on a log scale. The lines plot the mean across the instances, and the error bars represent 95% confidence intervals about the mean.

The results in Figure 5 exhibit an easy→hard→easy pattern. When we supply a cost bound much lower than the cost of an optimal solution, it is easy to show that the problems are unsolvable within C. As C approaches what we assume is the optimal solution cost the difficulty peaks. As the cost bound continues to rise past this critical point, problems once again become easy to solve. This parallels a similar phenomenon in constraint satisfaction problems except in an optimization setting.

In the left hand side of the peak in Figure 5, we can see that the algorithms appear indistinguishable from one another. This corresponds to small values of C, where most instances could not be solved. Figure 6 shows a closer view of these unsolvable configurations. On the y-axis of this plot, we present the number of duplicate states encountered during search. We see that BEES and PTS encounter relatively few duplicate states. BEES expands nodes in almost exactly A* order because it almost always predicts that all nodes have cost above C. The behavior of PTS does not deviate largely from that of BEES. We see that BEEPS and $\widehat{PTS}$ re-expand many more nodes than BEES and PTS. This aligns with Observation 1.

In Figure 5, when we reach the portion of the space where solutions are easy to find because the cost bound is generous, we see that BEES and BEEPS are about two orders of magnitude faster than PTS and are almost an order of magnitude faster than $\widehat{PTS}$. This supports our Observation 2, that for domains with non-uniform action costs, BEES and BEEPS were likely to be the best performing algorithms. In this domain, $\widehat{PTS}$ is also taking advantage of actions-to-go, albeit indirectly as it uses $\widehat{d}$ to calculate $\widehat{h}(n)$ ($\widehat{h}(n) = h(n) + \bar{\epsilon_h} \cdot \widehat{d}(n)$ where $\bar{\epsilon_h}$ is the mean one-step error in h), but it never explicitly orders nodes according to $\widehat{d}$. We see that the two algorithms that search directly on $\widehat{d}$ outperform the algorithm that uses it indirectly.

Sliding Tiles Puzzles We use the 100 instances of the 15-puzzle presented by Korf (1985). h is the Manhattan distance heuristic and $\widehat{h}$ was computed using the same online corrections as before. Because actions have unit cost, $\widehat{h}$ and $\widehat{d}$ are identical.

Results for the algorithms in this domain are presented in Figure 7 and Figure 8. Again we see an easy→hard→easy pattern. Remarkably, algorithms which were very different from one another in the previous domain are largely indistinguishable here — The confidence intervals overlap substantially. This domain has uniform-cost actions and so this supports Observation 3. The overhead of generating successor states and computing heuristics is very low in this domain, so PTS appears to have the best mean performance, in line with Observation 4. This is confirmed in Figure 8, which measures the number of nodes generated during search. The algorithms generate very similarly many nodes, indicating that the timing differences in Figure 7 can be attributed to the differing overheads in the algorithms.

Inverse Tiles We study the same 100 15-puzzle instances as above, but we replace the standard cost function with one where the cost of moving a tile is the inverse of its face value, $\frac{1}{face}$. This separates the cost and length of a solution without altering other properties of the domain, such as its connectivity and branching factor. This domain was first suggested by Thayer, Dionne, and Ruml (2011). h is computed as the weighted sum of Manhattan distances for each tile, that is $h(n) = \frac{MD(n)}{face(n)}$, and d is the unweighted Manhattan distance.

The performance of the algorithms in the inverse tiles problem is shown in Figure 9. We once more see the easy→hard→easy transition. As we saw in the vacuum domain, algorithms that take the varying costs of actions into account perform better. $\widehat{PTS}$ is consistently two orders of magnitude faster than PTS; recall that $\widehat{h}(n)$ relies on $d(n)$. BEES and BEEPS, which directly rely on search on $\widehat{d}$, are more than four orders of magnitude faster than PTS for many cost bounds, confirming Observation 2.

Bounded-Cost vs. Bounded Suboptimal Search Stern, Puzis, and Felner (2011) noted that bounded suboptimal search algorithms can be converted into bounded-cost search algorithms by tuning the supplied suboptimality bound so that it provides good performance for the desired cost bound. Figure 12 shows the performance of the state-of-the-art bounded-suboptimal search algorithm, EES, as a bounded-cost algorithm. Rather than attempting to tune the suboptimality bound for EES, we provide a wide range of suboptimality bounds to compare to the bounded-cost algorithms presented in this paper. The version of EES used here accepts two parameters: the suboptimality bound originally required by EES and a cost bound. It differs from the original version only in that it prunes any node with $f(n) > C$.

In this plot we see that no setting for EES outperforms BEES or BEEPS. This is because EES is designed for bounded suboptimal search, not bounded-cost search. While searching for a solution within a given cost bound, EES is also proving that the returned solution lies within the given suboptimality bound as well. Proving that a solution exists within a suboptimality bound requires expanding all nodes with an $w \cdot f(n) \leq g(inc)$, where inc is the incumbent solution returned by EES. For tight bounds (small w) or weak admissible heuristics, this can be a great many nodes.

EES doesn't behave identically to BEES or BEEPS for

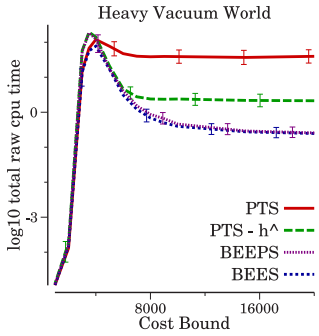

Figure 5

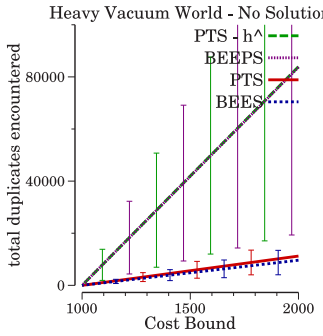

Figure 6

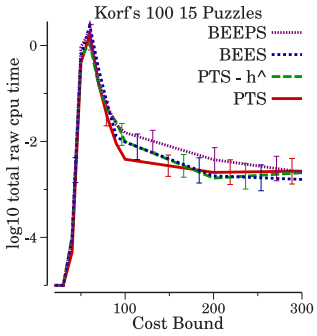

Figure 7

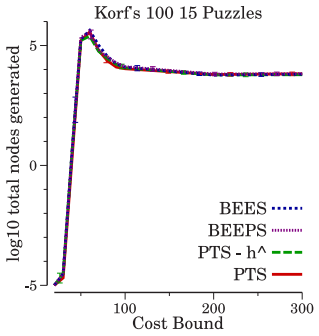

Figure 8

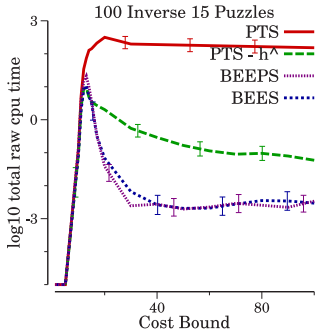

Figure 9

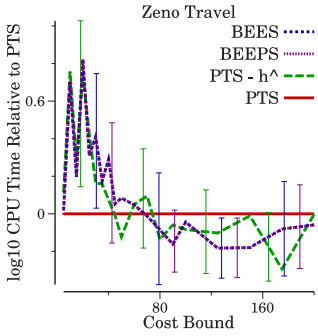

Figure 10

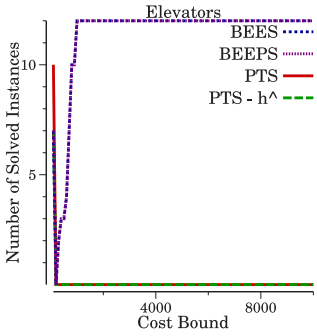

Figure 11

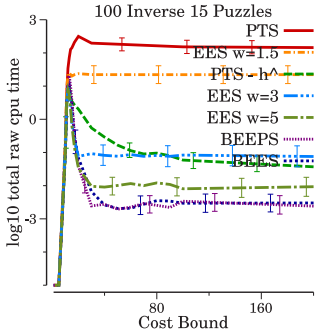

Figure 12

high suboptimality bounds. EES relies on a suboptimality bound to determine what nodes are likely to lie within a the bound. Specifically it considers all nodes with $\widehat{f}(n) \leq w \cdot \widehat{f}(best_{\widehat{f}})$. It is possible that many (even all) of these nodes have $\widehat{f}(n) > C$. Since EES can consider different nodes than BEES and BEEPS for the same cost bound, the nodes expanded may also differ.

Bounded-Cost vs. Baseline Algorithms There are three natural baselines for bounded-cost search: A* search with pruning, greedy search on cost-to-go with pruning, and greedy search on actions-to-go, often called speedy search, with pruning. In Figure 4 we compare the performance of these three natural baselines with the bounded-cost search algorithms evaluated in this paper. The y-axis is the number of nodes generated by a search on a long scale, the x-axis is the cost bound, also on a log scale. The performance of the A* baseline, the greedy baseline, and PTS are very similar. These three are all dominated by the speedy baseline. BEES, BEEPS, and $\widehat{PTS}$ all outperform the speedy baseline. For large C, the performance of BEES, BEEPS, and the speedy baseline converge as they are all performing best-first search on $\widehat{d}$.

To gain a deeper insight into the poor performance of PTS in the inverse tiles puzzle, consider Figure 3, which shows the relationship between the admissible heuristic and the true cost-to-go for the inverse tiles problem. Recall that PTS (when implemented with the f_{lnr} cost function), makes the assumption that h and h^* are linearly related. In the plot, we see that the relationship is not clearly linear. This may be contributing to the poor performance of PTS in this domain and highlights another shortcoming of PTS: PTS relies on how h is related to h^* which we may not know before solving an instance. This may be especially difficult when many different types of problem instances that will be solved. Consider pathfinding in a grid, where problems may have uniformly distributed obstacles, or be mazes, or be video-game maps with terrain of differing cost. Furthermore, if one does not know the domain of the problem in advance, relating h to h^* a priori is impossible. This is often the case in domain independent planing, the next setting we consider.

We chose to implement our algorithms in the FastDownward planning framework (Helmert 2006) because it is freely available and has been the basis of recent award-winning planners (Richter and Westphal 2010).

ZenoTravel In zenotravel, the goal is to move passengers from their point of origin to their chosen destination using ground transportation and airplanes. We use the 20 instances of the problem included in the FastDownward repository. For h, we used the LM-Cut heuristic, for $\widehat{h}$ we used the FF heuristic, and for $\widehat{d}$ we also used the FF heuristic ignoring action costs (*ff(cost_type=1)*). No special bookkeeping is done to reduce the cost of this calculation, we simply run the *FF* computation once with a unit-cost representation of the problem and once with the original problem. Extracting the number of actions in the relaxed plan is possible and would likely improve the timing results for BEES and BEEPS in the FastDownward planner.

Figure 10 shows the performance of algorithms. The y-axis shows the mean solving time relative to the time taken by PTS on a log scale. Unlike the previous domains, where

the instances formed a set of similarly difficult instances, the instances here span a wide range of difficulty. Normalizing to the performance of PTS allows us to compare across this diverse set.

PTS starts off stronger than the other algorithms. This is reasonable because few problems are solvable, so only the admissible cost-to-go heuristic is really useful as it is what allows algorithms to prune unpromising nodes. Furthermore, as mentioned above, $\widehat{h}$ and $\widehat{d}$ are computed using the FF heuristic. FF is known to be very effective in guiding the search to a goal, but it can also be a poor estimator of h^*. Consequently, for unsolvable instances BEES and BEEPS may often consider nodes as solvable, and will therefore expand nodes in a greedy manner. This will result in additional node reexpansions, and degraded performance, as explained in Observation 1. Furthermore, $\widehat{PTS}$, BEES, and BEEPS are all computing additional heuristics, which contribute very little to the search in this part of the plot. As the cost bound grows, more problems become solvable and computing the additional heuristics begins to pay off.

Elevators The elevators domain was one of the new non-unit-cost domains included in the 2011 international planning competition. The goal is to minimize the cost of transporting all passengers from their current floor to their desired floor, and elevators have differing speeds (some elevators are express elevators, some stop on every floor). We evaluated on the 20 instances used in the sequential satisficing track of the 2011 IPC.

Figure 11 shows the performance of the algorithms in the elevators domain. Rather than reporting mean solving time, we report the number of instances solved for a given cost bound. We see that algorithms that do not make any use of distance-to-go information, i.e. PTS and $\widehat{PTS}$, solve no instances for the majority of the cost bounds. For tight bounds where no solution exists, they do manage to solve some problems, but in the case where a solution exists and they must find it, they fail. Again, for very low C, PTS proves no solution exists for three more instances because it has reduced overhead (it computes fewer heuristics). BEES and BEEPS, on the other hand, can both prove that no solution exists below C for small values of C and also find solutions to the problem when C is large.

In conclusion, in accordance with Observation 2, in domains with non-uniform action costs, algorithms that rely on estimates of actions-to-go (i.e., BEES and BEEPS) find solutions faster than those that do not. In heavy vacuums and inverse tiles, using $\widehat{d}$ for guidance provided speedups of several orders of magnitude. For the elevators domain, BEES and BEEPS solve several problems. For all but the most conservative C values, PTS and $\widehat{PTS}$ solved no instances.

The speedup on domains with nonuniform-action costs did not come at the cost of performance on domains with uniform-cost actions, as we saw in the results for the standard tiles puzzle. While there were minor difference in timing, in terms of the number of states evaluated, the algorithms were indistinguishable. When we evaluated the algorithms on a unit-cost planning domain, we saw that algorithms that use inadmissible heuristics outperformed those that did not.

Discussion

In this paper, we introduced three novel bounded-cost search algorithms: $\widehat{PTS}$, Bounded-Cost Explicit Estimation Search (BEES) and Bounded-Cost Explicit Estimation Potential Search (BEEPS). In contrast to the previous bounded-cost search algorithm (PTS), both BEES and BEEPS consider a heuristic estimate of the number of actions between a given node and a goal (the $\widehat{d}$ function). In every iteration, the node with the lowest $\widehat{d}$ in expanded among the nodes that are estimated to lead to a bounded-cost goal. If no such nodes exists, BEES behaves like A* while BEEPS behaves like $\widehat{PTS}$. Both BEES and BEEPS were evaluated on five diverse domains, and show several orders of magnitude improvement over PTS in every domain where actions have non-uniform costs. Sometimes those differences were up to four orders of magnitude, but often they were at least a single order of magnitude.

BEES and BEEPS do not need to make assumptions about the relationship between the heuristics and truth, unlike PTS. This makes these algorithms more general. When PTS is used in a domain without a linear relationship between h and h^*, an appropriate potential-approximating function must be found or constructed, or PTS risks poor performance.

Both BEES and BEEPS use distance estimates to guide the search. However, distance estimates are merely a proxy for the values we really want to have. We don't want to minimize solution length subject to a cost bound, we want to minimize solving time. Solving time is directly related to search effort in terms of the number of nodes generated, which happens to correlate with the estimated number of actions to a solution (Dionne, Thayer, and Ruml 2011). Finding an efficient method for estimating remaining search effort is the subject of future work and would benefit not just bounded-cost search, but also bounded suboptimal search algorithms that make use of distance estimates as a proxy for search effort, such as EES (Thayer and Ruml 2011), and may also improve the performance of anytime search algorithms as well.

Conclusions

In this paper we showed how inadmissible cost and length estimates can be used to solve bounded-cost search problems efficiently. This reflects a general trend in suboptimal search, where we have seen bounded suboptimal search algorithms like EES and anytime solvers such as LAMA-11 benefit from making a distinction between the cost and length of a solution when searching. Suboptimal search algorithms can and should use additional information such as distance-to-go estimates and inadmissible heuristics, in addition to the traditional g and h values.

Acknowledgments

We graciously acknowledge funding from the DARPA CSSG program (grant HR0011-09-1-0021), NSF (grant IIS-0812141), and the Israeli Science Fund (grant 305/09).

References

Dionne, A.; Thayer, J. T.; and Ruml, W. 2011. Deadline aware search using online measures of behavior. In *The 2011 International Symposium on Combinatorial Search (SOCS-11)*.

Haslum, P., and Geffner, H. 2001. Heuristic planning with time and resources. In *European Conference an Planning (ECP*.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 191–246.

Helmert, M. 2011. Fast downward planner. `http://www.fast-downward.org`. Accessed Decemeber 16, 2011.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Jabbari Arfaee, S.; Zilles, S.; and Holte, R. C. 2011. Learning heuristic functions for large state spaces. *Artif. Intell.* 175(16-17):2075–2098.

Korf, R. E. 1985. Iterative-deepening-A*: An optimal admissible tree search. In *Proceedings of IJCAI-85*, 1034–1036.

Nakhost, H.; Hoffmann, J.; and Müller, M. 2010. Improving local search for resource-constrained planning. In *SOCS*.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Phillips, M., and Likhachev, M. 2011. Planning in domains with cost function dependent actions. In *AAAI*.

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

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Russell, S., and Norvig, P. 2010. *Artificial Intelligence: A Modern Approach*. Third edition.

Samadi, M.; Felner, A.; and Schaeffer, J. 2008. Learning from multiple heuristics. In *Proceedings of AAAI-08*.

Stern, R. T.; Puzis, R.; and Felner, A. 2011. Potential search: A bounded cost search algorithm. In *Proceedings of ICAPS-11*.

Thayer, J. T., and Ruml, W. 2011. Bounded suboptimal search: A direct approach using inadmissible estimates. In *Proceedings of IJCAI-11*.

Thayer, J. T.; Dionne, A.; and Ruml, W. 2011. Learning inadmissible heuristics during search. In *Proceedings of ICAPS-11*.

Thayer, J. T.; Ruml, W.; and Kreis, J. 2009. Using distance estimates in heuristic search: A re-evaluation. In *Symposium on Combinatorial Search*.