Journal of Artificial Intelligence Research 47 (2013) 697–740 Submitted 05/13; published 08/13

Heuristic Search When Time Matters

Ethan Burns EABURNS AT CS.UNH.EDU
Wheeler Ruml RUML AT CS.UNH.EDU
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA

Minh B. Do MINH.B.DO AT NASA.GOV
Planning and Scheduling Group
SGT Inc.
NASA Ames Research Center
Moffett Field, CA 94035 USA

Abstract

In many applications of shortest-path algorithms, it is impractical to find a provably optimal solution; one can only hope to achieve an appropriate balance between search time and solution cost that respects the user's preferences. Preferences come in many forms; we consider utility functions that linearly trade-off search time and solution cost. Many natural utility functions can be expressed in this form. For example, when solution cost represents the makespan of a plan, equally weighting search time and plan makespan minimizes the time from the arrival of a goal until it is achieved. Current state-of-the-art approaches to optimizing utility functions rely on anytime algorithms, and the use of extensive training data to compute a termination policy. We propose a more direct approach, called Bugsy, that incorporates the utility function directly into the search, obviating the need for a separate termination policy. We describe a new method based on off-line parameter tuning and a novel benchmark domain for planning under time pressure based on platform-style video games. We then present what we believe to be the first empirical study of applying anytime monitoring to heuristic search, and we compare it with our proposals. Our results suggest that the parameter tuning technique can give the best performance if a representative set of training instances is available. If not, then Bugsy is the algorithm of choice, as it performs well and does not require any off-line training. This work extends the tradition of research on metareasoning for search by illustrating the benefits of embedding lightweight reasoning about time into the search algorithm itself.

1. Introduction

Many problems in artificial intelligence can be formulated as shortest path problems, which can be solved using heuristic search algorithms such as A* (Hart, Nilsson, & Raphael, 1968). Unfortunately, because state spaces often grow exponentially with problem size, it is usually infeasible to find optimal solutions to shortest path problems of practical interest. Instead, practitioners tend to settle for suboptimal solutions, which can often be found more efficiently but will be more expensive to execute. One is left with the choice of spending a long time searching for a cheap solution, or a little time searching for an expensive one. We argue for a new approach that is not strictly concerned with optimizing solution cost, but with optimizing a utility function given in terms of both solution cost and search time. With such a utility function, a user can specify a preference between search time and solution cost, and the algorithm handles the rest.

We consider utility functions given as a linear combination of search time and solution cost. This is an important form of utility function for two reasons. First, it is easily elicited from a user if not already explicitly in their application domain. For example, if cost is given in monetary terms, it is usually possible to ask how much time one is willing spend to decrease the solution cost by a certain amount. Second, if the solution cost is given in terms of time (i.e., the cost represents the time required for the agent to execute the solution), then this form of utility function can be used to optimize what we call *goal achievement time*; by weighting search time and execution time equally, a utility-aware search will attempt to minimize the sum of the two, thus attempting to behave such that the agent will achieve its goal as quickly as possible.

Most existing techniques for this problem are based on anytime algorithms (Dean & Boddy, 1988), a general class of algorithms that emit a stream of solutions of decreasing cost until converging on an optimal one. With sufficient knowledge about the performance profile of an anytime algorithm, which represents the probability that it will decrease its solution cost by a certain amount given its current solution cost and additional search time, it is possible to create a stopping policy that is aware of the user's preference for trading solving time for solution cost (Hansen & Zilberstein, 2001; Finkelstein & Markovitch, 2001).

There are two disadvantages to using anytime algorithms to trade-off solving time for solution cost. The first is that the profile of the anytime algorithm must be learned off-line on a representative set of training instances. In many settings, such as domain-independent planning, the problem set is unknown, so one cannot easily assemble a representative training set. Also, it is often not obvious which parameters of a problem affect performance so it can be difficult to tell if a problem set is representative. Even if an instance generator is available, the instances that it generates may not represent those seen in the real world. The second issue is that, while the stopping policy is aware of the user's preference for time and cost, the underlying anytime algorithm is oblivious and will emit the same stream of solutions regardless of the desired trade-off. The policy must simply do the best that it can with the solutions that are found, and the algorithm may waste a lot of time finding many solutions that will simply be discarded. Only the search algorithm itself is fully aware of the possible candidate solutions that are available and their relative estimated merits.

This paper presents four main contributions. First, we combine anytime heuristic search with the dynamic programming-based monitoring technique of Hansen and Zilberstein (2001). To the best of our knowledge, we are the first to apply anytime monitoring to anytime heuristic search. Second, we present a very simple portfolio-based method that estimates a good parameter to use for a bounded-suboptimal search algorithm to optimize a given utility function. Third, we present Bugsy, a best-first search algorithm that does not rely on any off-line training, yet accounts for the user's preference between search time and solution cost.[1] One important difference between Bugsy and most previous proposals for trading-off deliberation time and solution cost is that Bugsy considers the trade-off directly in the search algorithm, whereas previous techniques, such as those based on anytime algorithms, only consider the trade-off externally to the actual search algorithm. Finally,

1. A previous version of Bugsy was proposed by Ruml and Do (2007), see Appendix A for a discussion of the improvements incorporated in the version presented here.

we present the results of a set of experiments comparing the portfolio-based method, anytime monitoring, and Bugsy, along with utility-oblivious algorithms such as A* and greedy best-first search, real-time search algorithms, and decision-theoretic A* (DTA*, Russell & EricWefald, 1991), a previously proposed utility-aware search. While there has been much work discussing the trade-off between deliberation and solution cost, to the best of our knowledge we are the first to implement and thoroughly evaluate many of these ideas in the context of heuristic search.

The results of our experiments reveal two surprises. First, if a representative set of training instances is available, the most effective approach is the very simple technique of selecting a bound to use for a bounded-suboptimal search. Surprisingly, this convincingly dominates anytime algorithms with monitoring in our tests. Second, neither Bugsy or anytime search with monitoring dominates the other. Bugsy does not require any off-line training, yet surprisingly, Bugsy can perform as well as the methods that use training data. If a representative problem set is not available, then Bugsy is the algorithm of choice. This work extends the tradition of research on metareasoning for planning by illustrating the benefits of embedding lightweight reasoning about time into the search algorithm itself.

2. Background

In this section we briefly describe heuristic search, present some terminology used in the remainder of this paper, and discuss the type of utility functions we are addressing.

2.1 Heuristic Search

As considered in this paper, heuristic search is a technique for finding the shortest path between nodes in a weighted graph; many problems can be specified in this form. Since it is typical for these graphs to be much too large to represent explicitly, algorithms usually generate the graph lazily using a function called *expand*. The expand function returns the successors of a node in the graph. We call the process of evaluating the expand function on a node *expanding* the node, and when expanding a node we say that we are *generating* its successors.

A* (Hart et al., 1968) is probably the best-known heuristic search algorithm. It maintains two sets of nodes: the *open list* contains the frontier nodes that have been generated but not yet expanded, and the *closed list* contains nodes that have already been expanded (a common optimization is is for the closed list to also include nodes that are already on the open list too), and therefore represent duplicate states if encountered again. The open list is sorted on $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the path from the initial node to node n, and $h(n)$ is the *heuristic* estimate of the cheapest path cost from n to any goal node reachable from n. The algorithm proceeds by removing a node with the minimum f value from the open list, expanding it, putting its children on the open list, and putting the node on the closed list. If A* removes a goal node from its open list, then it stops searching and returns the path to the goal as the solution. Finally, if the heuristic never over-estimates the cost to go then it is called *admissible*. With an admissible heuristic, A* returns optimal solutions.

Dechter and Pearl (1988) prove that if the heuristic satisfies a property called *consistency* (for all nodes n and m, $h(n) \leq h(m) + c(n, m)$, where $c(n, m)$ is the cost of the cheapest path between n and m), then A* expands the fewest possible nodes required to prove the optimality of its solution with the given heuristic. In practice A* often takes too long (Helmert & Röger, 2008), thus given its optimal efficiency it is infeasible to look for optimal solutions to many problems. Instead, one must settle for suboptimal solutions, with the hope that it is possible to find a sufficiently cheap solution within a reasonable amount of time and memory.

2.2 Suboptimal Search

Greedy best-first search (Michie & Ross, 1969) is a popular suboptimal search algorithm. It proceeds like A*, but orders its open list only on the heuristic, $h(n)$, with the idea that remaining search effort correlates with remaining solution cost. In other words, it assumes that it will be easier to find a path to the goal from nodes with low h. When strictly attempting to minimize search time, Thayer and Ruml (2009) show that greedy best-first search on a different heuristic, d, can be more effective. Instead of estimating cost to go, as is done by traditional h functions, the d heuristic, called a *distance estimate*, estimates the number of remaining search nodes on the path to the cheapest solution beneath a node. In practice, distance estimates are as readily available as cost-to-go heuristics and can provide much better performance when used in greedy best-first search on domains where less cost to go is not directly correlated with less search to go. We call greedy best-first search using the d heuristic *Speedy* search, in analogy to greedy search.

While greedy best-first search can find solutions very quickly, there is no bound on the cost of its solutions. Bounded-suboptimal search algorithms remedy this problem. Weighted A* (Pohl, 1970) is perhaps the most common of these techniques—it proceeds just like A*, but it orders the open list on $f'(n) = g(n) + w \cdot h(n)$, with $w \geq 1$. The weighting parameter, w, puts more emphasis on the heuristic estimate than the cost of arriving at a node, thus it is greedier than A* and it often finds suboptimal solutions much faster than A* finds optimal ones. In addition, the weight provides a bound on the suboptimality of its solutions: the solutions are no more than w times the cost of the optimal solution (Pohl, 1970). Unlike greedy best-first search, weighted A* lets the user select a weight, allowing it to provide either cheaper solutions or faster solutions depending on their needs.

We refer the reader to the work of Thayer (2012) for a more in-depth study of suboptimal and bounded-suboptimal search algorithms, including many that use d heuristics.

2.3 Utility Functions

So far, we have described A*, which optimizes solution cost, bounded-suboptimal search, which finds solutions within a constant factor of optimal, and greedy best-first search, which attempts to minimize solver time. Often, none of these are really desired: optimal solutions require an impractical amount of resources, one rarely requires solutions strictly within a given bound of optimal, and unboundedly suboptimal solutions are too costly. Instead, we propose optimizing a simple utility function given as a linear combination of search time and solution cost:

$$U(s, t) = -(w_f \cdot g^*(s) + w_t \cdot t) \tag{1}$$

where s is a solution, $g^*(s)$ is the cost of the solution, t is the time at which the solution is returned, w_f and w_t are user-specified weights used to express a preference for trading-off search time and solution cost. The number of time units that the user is willing to spend to achieve an improvement of one cost unit is w_f/w_t. This quantity is usually easily elicited from users if it is not already explicit in the application domain. The cost of the empty solution, $g^*(\{\})$, is a user-specified value that defines the utility achieved in the case that the search gives up without returning a solution

A linear utility function has two main benefits. First, they are fairly expressive. For example, one can optimize for cost if the both the solution and search time are given in monetary terms. This situation can occur in cloud computing environments where computation time costs money. A linear utility function can also capture optimal or greedy search by using 0 for the weight on execution time and solution cost respectively. Additionally, a linear utility function can express goal achievement time by weighting search time equally with solution makespan. Some practical examples where minimizing goal achievement time is desired include robotic and video game pathfinding problems. In these settings, a user often does not care about optimal solutions if they take too long to find, they may only care about achieving the goal as quickly as possible.

As a demonstration of minimizing goal achievement time, we have made a video of A*, Speedy search, and Bugsy solving a pathfinding video game pathfinding problem. It is available in the online appendix of this paper or on the web: `http://youtu.be/Yluf88V1PLU`. The video includes three panels, each showing an agent using a different search algorithm. Since they do not focus on finding cost-optimal solutions, both the Speedy and Bugsy agents begin moving almost immediately. The A* agent stands still for a long time while it plans an optimal path, and it doesn't start moving until after Bugsy has arrived at the goal. While all of this is occurring, the Speedy agent is following an extremely circuitous path; it doesn't reach the goal until approximately 30 seconds after A*. We didn't show these agonizing seconds in the video, and instead stopped the recording as soon as A* reached the goal. Clearly, the Bugsy agent, which optimizes goal achievement time, not solution cost or search time, is preferred in this scenario.

While quite expressive, linear utility functions are also rather simple. One main benefit of the simplicity is that, with a fixed utility function, the passage of time decays all utility values at the same rate. This simplification allows us to ignore all time that has passed before the current decision point. We can then express utility values in terms of the utility of each outcome starting at the current moment in time. Without this benefit, the mere passage of time would change the relative ordering between the utilities of different outcomes; we would need to re-compute all utility values at every point in time in order to select the best outcome.

We only consider linear utility functions in this work, but it should be noted that one could consider other more expressive functions. Step functions, for example, can represent deadlines where after a certain amount of time has elapsed the utility of acting greatly decreases. Bugsy does not support such functions, but the anytime monitoring technique discussed in Section 3.1 has no restrictions on the utility functions that it can optimize. Anytime monitoring can naturally handle more expressive functions, like step functions.

3. Previous Work

Next we describe some previous techniques for trading-off solver time for solution cost.

3.1 Monitoring Anytime Algorithms

Much previous work in optimizing utility functions of solving time and cost, such as Equation 1, has focused on finding stopping policies for *anytime algorithms*. Anytime algorithms (Dean & Boddy, 1988) are a general class of algorithms that find not one solution, but a stream of solutions with strictly decreasing cost. They get their name because one can stop an anytime algorithm at any time to get its current best solution. Anytime algorithms are an attractive candidate for optimizing a utility function: since there is more than just a single solution from which to pick, there is more opportunity to choose a solution with a greater utility than when using an algorithm that just finds a single solution. Different solutions will be found at different times, and if we knew the time at which the algorithm would find each of its solutions and the cost of those solutions, then we could compute their utilities and return the solution that maximizes utility. Unfortunately, it is usually not possible to know what solutions an anytime algorithm will return without running it. Instead, while the algorithm is running, one must continually make the decision: stop now, or keep going?

Deciding when to stop is no easy task, because the utility of a solution depends not only on its cost but also on the time needed to find it. On one hand, stopping early can reduce the amount of computation time at the expense of having a more costly solution. On the other hand, if the algorithm continues, it may not reduce the solution cost by enough to justify the extra computation time. In this case, the final utility can be worse than it would have been had the algorithm stopped earlier. With a little extra information, however, it is possible to create a reasonable policy.

The Near Optimal Response-Time Algorithm (NORA, Shekhar & Dutta, 1989) provides one very simple stopping policy for optimizing goal achievement time. NORA, simply stops the anytime algorithm when the current search time is a user-specified factor of the current incumbent solution's execution time. Shekhar and Dutta (1989) prove that the, if the search stops when the time is a factor λ of the incumbent solution cost, then the goal achievement time will be within a factor of $\min(1 + \lambda, 1 + \frac{1}{\lambda})$ of the optimal goal achievement time.

Our use of NORA is slightly different than that of Shekhar and Dutta (1989). They did not apply NORA to anytime heuristic search. Instead, they evaluated it empirically on database query optimization problems, which are tree search problems, where every leaf node is a possible solution. They also describe how one could use NORA in an A* search, but they make the assumption that if A* is stopped early without reaching the goal then a heuristic planning procedure can be used to achieve the goal after executing the partial solution found by A*. Such a procedure is often not available. When using NORA with anytime heuristic search, as we do here, each incumbent solution is guaranteed to reach the goal. The only disadvantage is that, as with all anytime stopping policies, it cannot do better than the best solution found by the utility-oblivious anytime algorithm.

NORA finds a solution within a specified bound on the optimal goal achievement time. Instead, Hansen and Zilberstein (2001) present a dynamic programming-based technique for building an optimal stopping policy for any utility function. It requires one extra piece of information: the *profile* of the anytime algorithm. Hansen and Zilberstein define the profile as a probability distribution over the cost of the solution returned by the algorithm, conditioned on its current solution cost and the additional time it is given to improve its solution: $P(q_j|q_i, \Delta t)$, where q_j and q_i are two possible solution costs and Δt is the additional time. The profile allows for reasoning about how the solution cost may decrease if the algorithm is given more time to improve it. While this requires extra knowledge, we performed a small experiment (not shown here) and found that the optimal policy found using dynamic programming performs better than the simpler NORA technique.

Hansen and Zilberstein's technique monitors the progress of the anytime algorithm by evaluating the stopping policy at discrete time intervals. If the algorithm considers stopping every Δt time units, then the utility achievable at time t when the algorithm's current solution costs q_i is:

$$V(q_i, t) = \max_d \begin{cases} U(q_i, t) & \text{if d = stop,} \\ \sum_j P(q_j|q_i, \Delta t) V(q_j, t + \Delta t) & \text{if d = continue} \end{cases} \tag{2}$$

and the stopping policy is:

$$\pi(q_i, t) = \operatorname*{argmax}_d \begin{cases} U(q_i, t) & \text{if d = stop,} \\ \sum_j P(q_j|q_i, \Delta t) V(q_j, t + \Delta t) & \text{if d = continue} \end{cases} \tag{3}$$

where U is the user-specified utility function and P is the profile of the anytime algorithm. They also show a more sophisticated technique that accounts for the cost of evaluating the policy, however, for the algorithms presented in this paper, the cost of evaluating the policy consists of a mere array lookup and is essentially free.

Since the profile of an anytime algorithm is usually not known, it must be estimated. It is possible to estimate the profile off-line if one has access to a representative set of training instances. To estimate the profile, the algorithm can be run on each of the training instances and a 3-dimensional histogram can be created to represent the conditional probability distribution, $P(q_j|q_i, \Delta t)$, needed to compute the stopping policy (cf. Equation 3). Appendix C gives a more detailed description of our implementation of this procedure.

3.2 Anytime Heuristic Search

Anytime algorithms are a very general class and there are many anytime algorithms for heuristic search (Likhachev, Gordon, & Thrun, 2003; Hansen & Zhou, 2007; Richter, Thayer, & Ruml, 2010; van den Berg, Shah, Huang, & Goldberg, 2011; Thayer, Benton, & Helmert, 2012). In this paper we use Anytime Repairing A* (ARA*, Likhachev et al., 2003) since it tended to give the best performance over other approaches according to experiments done by Thayer and Ruml (2010). ARA* executes a series of weighted A* searches, each with a smaller weight than the previous. Since the weight bounds the solution cost, the looser bounds on early iterations tend to find costly solutions quickly. As time passes and the weight decreases, so does solution cost, eventually converging to optimal. ARA* also has special handling for duplicates that are encountered during search that enables it to be more efficient while still guaranteeing a bound on each of its solutions.

Like most anytime heuristic search algorithms, ARA* has parameters. Before running ARA*, the user must select the weight schedule, which is typically comprised of an initial weight and the amount by which to decrement the weight after each solution is found. The behavior of ARA* varies with different weight schedules. For our experiments, we used an initial weight of 3.0 and a decrement of 0.02. This schedule was used by Likhachev

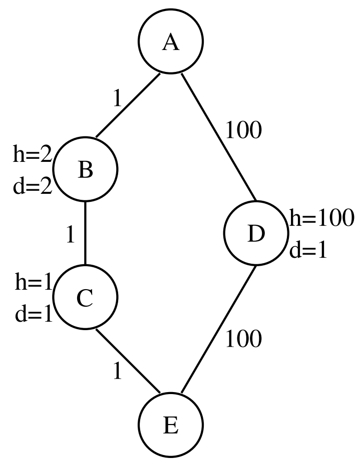

Figure 1: A small example graph.

et al. (2003), and we found that it gave the best performance when compared to several alternative schedules on the domains we considered.

Given a fixed weight schedule, an anytime heuristic search algorithm will emit a fixed stream of solutions for a given problem instance; the algorithm does not take the user's utility function into account. The same solutions will be found regardless of whether the user wants any solution as fast as possible or the optimal solution at all costs. Figure 1 shows a small, concrete example, where the goal is to find a path from node A to node E. Each node is labelled with it's heuristic value ("h") and the number of nodes remaining to the goal ("d"), and edges are labelled with their costs. If the user wants an optimal solution, then the algorithm would ideally return the path A, B, C, E. However, if the user wants any solution as fast as possible, then it may be better to find the solution A, D, E, as it has fewer nodes, and may be found in fewer expansions. ARA* only considers cost, not distance, so with an initial weight less than $66\frac{2}{3}$, the longer, cheaper, solution will be found regardless of the user's preference. It is up to the monitoring technique to select the best that it can from the solutions that are found.

3.3 Contract Search

Dionne, Thayer, and Ruml (2011) consider the problem of *contract search*, where a goal must be returned by a hard deadline. Unlike real-time search (Korf, 1990), where only the agent's next action must be ready by the deadline, contract search requires the algorithm to return a complete path to a goal. Like optimizing a utility function, contract search must be aware not only of the cost of solutions but also of the amount of time required to find them. While the conventional approaches to contract search use anytime algorithms, Dionne et al. (2011) present Deadline-Aware Search (DAS) which considers search time directly.

The basic idea behind DAS is to consider only states that lead to solutions deemed reachable within the deadline. Two different estimates are used to determine this set of nodes: an estimate of the maximum-length solution path that the search can explore before the deadline arrives, called d_{max}, and an estimate of the distance to the solution beneath each search node on the open list, in other words d. States for which $d \leq d_{max}$, are deemed reachable, all other states are "pruned." The search expands non-pruned nodes in best-first order on $f = g + h$, updating its d_{max} and d estimates on-line. If the updates cause all remaining nodes to be pruned while there is remaining time before the deadline, DAS uses a recovery mechanism to repopulate the open list from the set of pruned nodes and continues searching until the deadline is reached.

As mentioned previously, d estimates are as readily available as normal cost-to-go heuristics, h, for most domains. This leaves the question of how to estimate d_{max}. Dionne et al. (2011) show that simply using the remaining number of possible expansions, computed via the expansion rate and remaining time, is not appropriate due to a phenomenon that they call *search vacillation*. When a best-first search expands nodes, it typically does not expand straight down a single solution path, instead it considers multiple solution paths at the same time, expanding some nodes from each. When it does this, it is said to be *vacillating* between many different paths, and it may not return to work on a particular path until it has performed many expansions along others. To account for vacillation, Dionne et al. introduce a metric called *expansion delay* that estimates the number of additional expansions performed by a search between the expansion of two successive nodes along a single path. They define $d_{max} = \frac{t_{rem} \cdot t_{exp}}{delay}$, where t_{rem} is the time remaining before the deadline, t_{exp} is the average expansion rate, and *delay* is the average expansion delay. They compute the average expansion delay by averaging the difference in the algorithm's total expansion count between when each node is expanded and when it was generated.

Dionne et al. (2011) showed experimentally that DAS performs favorably to anytime-based approaches and alternative contract search algorithms, indicating that an approach that directly considers search time may also be beneficial for utility function optimization.

4. Off-line Bound Selection

We now turn to the first of the two new methods introduced in this paper.

In this section, we will present a very simple technique for trading search time for solution cost that is based on bounded-suboptimal search. Recall that bounded-suboptimal search algorithms return solutions that are guaranteed to be within a user-specified factor of the optimal solution cost. In practice, few applications require an actual bound, instead the bound is used by practitioners as a parameter that can be tweaked to speed-up the search if it is not finding solutions quickly enough. The fact that the bound can trade search time for solution cost makes it a prime candidate for automatic parameter tuning (Rice, 1976). That is exactly what we propose.

As with the anytime methods discussed in the previous section, off-line bound selection requires a representative set of training instances. The instances are used to gather information about how a bounded-suboptimal search trades-off search time for solution cost. The only other requirement is that the user select a set of diverse bounds to try as parameters to the search algorithm. The algorithm is then run on each of the N training instances with each suboptimality bound, creating a list of N pairs for each bound: $sols_b = \langle (c_1, t_1), ..., (c_N, t_N) \rangle$ where b is the bound passed as a parameter to the algorithm, c_i is the cost of the solution to the ith training instance and t_i is the time at which the ith solution was found. Given a utility function $U : cost \times time \to \mathbb{R}$, we can select the bound that gives the greatest expected utility on the training set:

$$bound_U = \underset{b}{\operatorname{argmax}} \left(\frac{1}{|sols_b|} \cdot \sum_{(c,t) \in sols_b} U(c,t) \right) \tag{4}$$

In our experiments, we select a different weight to use for each utility function from the set 1.1, 1.5, 2, 2.5, 3, 4, 6, and 10. It may be possible to reduce the number of weights in the training set by using linear interpolation to estimate the performance of parameters between those used for training. This simple approach can also be extended to select over a portfolio of different algorithms in addition to different bounds. It may be beneficial, for example, to include both A* and Speedy search in the portfolio, as these algorithms will likely be selected if cheap solutions are required or if a solution must be found very quickly. We will see in Section 6 that this very simple technique outperforms ARA* using an anytime monitor in our experimental evaluation. In fact, if a representative set of training instances is available, then this technique tends to perform better than all other algorithms that we evaluate.

A related technique is the dove-tailing method of Valenzano, Sturtevant, Schaeffer, Buro, and Kishimoto (2010). Their approach is presented as a way of side-stepping the need for parameter tuning by running all parameter settings simultaneously. They found that, with dove-tailing, weighted IDA* (Korf, 1985) was able to return its first solution much faster, as the dove-tailing greatly reduced the high variance in solving times for any given weight. They also found that dove-tailing over different operator orderings was effective for IDA*. The main difference between the work by Valenzano et al. and ours is that we have quite different goals. Our concern is not to find the first solution more quickly, but rather to select a setting that better optimizes a user-specified utility function. As such, our approach does not run multiple settings at the same time and instead selects a single parameter to run in a single search. In fact, the approaches are complementary. Given any of our utility-aware algorithms that have parameters, one could use dove-tailing to avoid the need to perform offline parameter selection.

5. Best-First Utility-Guided Search

Anytime search is not aware of utility. Monitoring and bound selection require training. In this section, we present Bugsy[2], a utility-aware search algorithm that does not require any off-line training.

5.1 Expansion Order

Like A*, Bugsy is a best-first search, but instead of ordering its open list on f, Bugsy orders its open list on an estimate of the utility of the outcome resulting from each node expansion. Since utility is dependent on time, the mere passage of time affects the utility values. This differs from most traditional search algorithms where the values used to order expansions remain constant. Recall, however, that when using a linear utility function, all utility values decay at the exact same rate. Given this, Bugsy ignores all past time

2. Bugsy is an acronym for ”Best-first Utility-Guided Search—Yes!”

and compares the utility estimates assuming that time begins at the current decision point. While these utility values will not match the utility of the ultimate outcome, they still preserve relative order of the different choices that the agent can make.

To understand BUGSY's ordering function, we will first consider the best utility of the outcome resulting from each node expansion as computed by an oracle. If we had foreknowledge of a maximum utility outcome, the only purpose of the search algorithm would be to achieve it by expanding the nodes along the path from the initial node in order to build the solution path. Since our utility function is given as a linear combination of solution cost and search time, the utility value of this outcome can be written in terms of the cost and length of a (possibly empty) maximum utility outcome, s:

$$U^* = -(w_f \cdot g^*(s) + w_t \cdot d^*(s) \cdot t_{exp}) \tag{5}$$

where $g^*(s)$ is the cost of the path s (recall that the cost of the empty path is a user-specified constant), $d^*(s)$ is the number of nodes on s, and t_{exp} is the time required to expand a node.[3]

Given the maximum utility value U^*, the best utility of the outcome resulting from expanding a node n is:

$$u^*(n) = \begin{cases} U^* & \text{if } n \text{ leads to a maximum utility outcome} \\ U^* - w_t \cdot t_{exp} & \text{otherwise} \end{cases} \tag{6}$$

In other words, the utility we get from expanding a node that leads to a maximum utility outcome is the maximum utility; expanding any other node is simply a waste of time, and has a utility of the maximum utility minus the cost of performing the unnecessary expansion.

In practice, we do not know the maximum utility, so we must rely on estimates. BUGSY uses two estimates to approximate the maximum utility. First, it estimates the cost of the solution that it will find beneath each node as, f. Note that f is an estimate, not only because the heuristic is an estimate of the true cost to go, but also because the cheapest solution beneath a node may not be the solution of greatest utility. See Appendix A for possible alternatives. Second it estimates the number of expansions required to find a solution beneath each node n, $exp(n)$. One crude estimate for remaining expansions is d, the distance heuristic that estimates the remaining nodes on the solution path. In reality, BUGSY will experience search vacillation, as discussed earlier, expanding more nodes than just those along a single solution path. To account for this vacillation, we use the expansion delay technique of Dionne et al. (2011) and we estimate $exp(n) = delay \cdot d(n)$. That is, we expect each of the remaining $d(n)$ steps to a goal will require *delay* expansions.

BUGSY can either choose to expand a node, or it can stop and return the empty solution. This is one way in which BUGSY differs from A*: BUGSY decides among actions at the search level (such as terminating the search, or expanding one of the many open nodes), whereas A* is committed to expanding nodes in a fixed order. In BUGSY each node on the open list represents a possible outcome, so BUGSY's maximum utility can be estimated using the maximum of the utility estimates of all open nodes and Equation 5:

$$\hat{U} = \max \left\{ \max_{n \in open} -(w_f \cdot f(n) + w_t \cdot d(n) \cdot delay \cdot t_{exp}), U(\{\}, 0) \right\} \tag{7}$$

3. Note that expansion time is not constant in general, because it includes time to add and remove elements from data structures like the open list.

Bugsy(*initial*, $u(\cdot)$)
1. *open* $\leftarrow$ {*initial*}, *closed* $\leftarrow$ {}
2. do
3. $\quad n \leftarrow$ remove node from *open* with highest $u(n)$ value
4. $\quad$ if n is a goal, return it
5. $\quad$ add n to *closed*
6. $\quad$ for each of n's children c,
7. $\qquad$ if c is not a goal and $u(c) < 0$ or an old version of c is in *open* or *closed*
8. $\qquad\quad$ skip c
9. $\qquad$ else add c to *open*
10. $\quad$ if the expansion count is a power of two
11. $\qquad$ re-compute $u(n)$ for all nodes on the open list using the most recent estimates
12. $\qquad$ re-heapify the open list
13. loop to step 3

Figure 2: Pseudo-code for Bugsy.

Once the estimate $\hat{U}$ is found, it would be possible to substitute it for U^* in Equation 6 to estimate $u^*(n)$, the utility of the outcome from expanding each node on the open list. However, Bugsy is only going to expand one node, so there is no need to estimate $u^*(n)$ for each open node; Bugsy simply expands the node with the best estimated outcome. Additionally, instead of computing the maximization in Equation 7 from scratch each time it is about to expand a node, Bugsy simply orders its open list on $u(n) = -(w_f \cdot f(n) + w_t \cdot d(n) \cdot \mathit{delay} \cdot t_{\mathit{exp}})$, each iteration popping off the node with the maximum $u(n)$ for expansion. In this way, the algorithm directly attempts to maximize utility.

Recall Figure 1, which shows two paths from an initial node, A, to a goal node, E. Because Bugsy accounts for distance in its utility function, it will find the shorter path A, D, E if their utility function sufficiently emphasizes finding solutions quickly over finding cheaper solutions. On the other hand, if the utility function gives a preference to finding cheap solutions then Bugsy will spend the extra search time to find the cheaper path, A, B, C, E.

5.2 Implementation

Figure 2 shows high-level pseudo-code for Bugsy. For clarity, the code elides the details of computing $u(n)$ values. The algorithm proceeds like A*, selecting the open node with the highest $u(n)$ for expansion (line 3). If this node is a goal, then it is returned as the solution (line 4), otherwise the node is put on the closed list (line 5) and its children are generated. Each new child is put onto the open list (line 9) except duplicate nodes and nodes for which expansion is estimated to have a negative utility (which occurs when the utility of returning no solution is greater than that of continuing the search); these are discarded (lines 7–8). Bugsy estimates the current expansion time and the expansion delay online, and these estimates can change after each expansion. Instead of re-sorting the open list after each expansion, Bugsy re-sorts whenever the number of nodes that it has expanded is a power of two, the utility of each open node is re-computed using the latest set of estimates for t_{exp} and expansion delay (as described in Section 3.3), and the open list is re-heapified (lines 10–12). We describe this re-sorting step in greater detail in Section 5.5.

5.3 Stopping

Bugsy orders its open list in decreasing order of $u(n)$, and stops searching when the maximum estimated utility is less than that of returning the empty solution. While it may be possible to continue searching in an anytime fashion after the first goal is found, from a utility perspective this is not the correct approach. We prove that here:

Theorem 1 *Assuming the expansion time t_{exp} is constant, h is admissible, and exp never overestimates the expansions to go, at the time that* Bugsy *finds its first solution, s, the solutions* Bugsy *would find beneath the remaining nodes would result in less utility than immediately returning s.*

Proof: Let $\mathcal{T}$ be the current time at which Bugsy found solution s. The utility of returning s is $U(s, \mathcal{T}) = u^*(s) = -(w_f \cdot f^*(s) + w_t \cdot \mathcal{T})$, where $u^*(s)$ is the utility of returning s now, and $f^*(s)$ is the cost of solution s. Note that because h is admissible and s is a goal, $h(s) = 0$, $g(s) = g^*(s)$, $f(s) = f^*(s)$, and therefore $u(s) = u^*(s)$. Also exp never overestimates the expansions to go so $exp(s) = 0$. Since s was chosen for expansion $u(n) \leq u^*(s)$ for every node n on the open list.

Let $t(n)$ be the minimum amount of additional time Bugsy requires to find the solution beneath any unexpanded node n. $t(n) \geq t_{exp}$ since Bugsy must at least expand n. So for each node n on the open list, the best utility that Bugsy could achieve by going straight to the cheapest goal under n is:

$$\begin{aligned}
u^*(n) &= -(w_f \cdot f^*(n) + w_t \cdot (t(n) + \mathcal{T})) \\
&\leq -(w_f \cdot f(n) + w_t \cdot (t(n) + \mathcal{T})), \text{ since } f(n) \leq f^*(n) \text{ due to the admissibility of } h \\
&\leq -(w_f \cdot f(n) + w_t \cdot (exp(n) \cdot t_{exp} + \mathcal{T})), \text{ since } exp \text{ never overestimates} \\
&= u(n), \text{ by the definition of } u(n) \\
&\leq u^*(s), \text{ since } u^*(s) = u(s) \text{ and } s \text{ was chosen for expansion, not } n
\end{aligned}$$

□

This justifies Bugsy's strategy of returning the first goal node that it selects for expansion. It should be noted that Bugsy's estimate of $exp(n) = delay \cdot d(n)$ is not a lower bound, but as we will see in the later sections, this stopping criterion performs quite well in practice.

5.4 Heuristic Corrections

Many best-first search algorithms use admissible heuristic estimates that never overestimate the true cost to go. The proof of optimality of A* and the proofs of bounded suboptimality of bounded suboptimal search algorithms rely crucially on the admissibility property of the heuristic. Bugsy does not fixate on cost-optimal solutions and does not guarantee bounded cost. Instead, Bugsy attempts to optimize a utility function for which solution cost is only one of two terms. Since there are no strict cost guarantees, Bugsy is free to drop the admissibility requirement if more informed but inadmissible estimates are available.

Thayer, Dionne, and Ruml (2011) show that inadmissible estimates can provide better performance for bounded suboptimal search. One such technique attempts to correct the heuristic estimates on-line using the average single-step error between the heuristic values of a node and its best child. Thayer et al. show that while this technique provides good search guidance, it is actually less accurate at estimating the true cost-to-go values than the standard admissible heuristics. For Bugsy, this is undesirable, as it does not need good guidance, but proper estimates. Thayer et al. also show that learning the heuristic off-line with linear regression can provide more accurate estimates. Unfortunately, using such off-line training would negate one of Bugsy's main benefits. It is a matter of empirical evaluation as to whether any of these techniques will provide better performance for Bugsy. In Section 6.5, we show that using the standard admissible heuristics often gives the best performance anyway.

5.5 Resorting

Instead of requiring off-line training as in the previous approaches, Bugsy uses on-line estimates to order nodes on its open list. First, while many analyses regard t_{exp} as as a constant, it can in practice depend on log-time heaps, cache behavior, and multiprogramming overhead, among other factors, so our implementation of Bugsy estimates t_{exp} as a global average computed during search. Second, Bugsy's expansion delay estimate is calculated as the global average of the difference in expansion count from when each node was generated to when it was expanded; this too must be done on-line. Unfortunately, the on-line estimates may change at each node expansion, and naïvely using the latest estimates to compute the u value for newly generated nodes can lead to poor performance. This is due to the comparisons used to order the open list: instead of fair comparisons based on the estimated utility of each node, the recent and very fresh estimates of new nodes will be compared with the old and possibly more stale estimates of nodes that have been open for a long time.

To alleviate this problem, our implementation of Bugsy uses two sets of estimates: one stable set used to order the open list, and one ever-changing set maintaining the most recent estimates. At certain points throughout the search, Bugsy copies the most up-to-date estimates into the stable set, recomputes the utility values of all open nodes, and re-sorts the open list. Our open list is implemented as a binary heap so it can re-establish the heap property in linear time in the number of elements on the heap. Unfortunately, it would still be very expensive to do this at every node expansion, so, instead, Bugsy reorders the open list exponentially less frequently as the search progresses—it only reorders when the number of expansions is a power of two. We prove that this *logarithmic scheme* only adds a constant amount of overhead per-expansion when amortized over the entire search.

Theorem 2 *In a search space that grows geometrically with a finite branching factor, the overhead of reordering the open list on power-of-two expansions is constant for each expansion when amortized over the search.*

Proof: Let b be the maximum branching factor. The maximum number of nodes that can be on the open list after n expansions is $N(n) = bn - n = n(b-1)$. The total cost of all re-sorting after n expansions is no more than:

$$\begin{aligned}
\sum_{i=0}^{\lfloor \lg n \rfloor} \mathcal{O}(N(2^i)) &= \sum_{i=0}^{\lfloor \lg n \rfloor} \mathcal{O}(2^i \cdot (b-1)), \text{ by definition of } N \\
&= c(b-1) \sum_{i=0}^{\lfloor \lg n \rfloor} 2^i, \text{ for some } c > 0, \text{ by definition of } \mathcal{O} \\
&= c(b-1)(2^{\lfloor \lg n \rfloor + 1} - 1), \text{ by the identity } \textstyle\sum_{i=0}^{j} 2^i = 2^{j+1} - 1 \\
&\leq c(b-1)(2 \cdot 2^{\lg n} - 1) \\
&= c(b-1)(2n-1) \\
&= \mathcal{O}(n)
\end{aligned}$$

□

So, the overhead per-expansion is constant when amortized over all expansions. It is a matter of empirical evaluation to determine if this constant overhead is detrimental—we address this in Section 6.4.

6. Experimental Evaluation

All the techniques discussed above involve approximations and estimations that may or may not work well in practice. In this section, we present results of an experimental comparison of the techniques to better understand their performance. All of these algorithms and domains were implemented in C++; the source code is available at `https://github.com/eaburns/search`.

6.1 Overview

In the following sections, we answer several questions experimentally. First, we would like to ensure that our monitored ARA* algorithm is performing at its best by comparing the profile learned off-line with an oracle. As we will see, the off-line profile, while only an estimate of the true profile of the algorithm, is quite well-informed.

In Section 5.5, we proved that re-sorting only adds a constant overhead per-expansion when amortized over the entire search. It is a matter of empirical evaluation to determine whether or not the benefits outweigh this overhead. Our experiments show that re-sorting with a logarithmic schedule greatly outperforms Bugsy without re-sorting.

In Section 5.4 we pointed out that Bugsy does not require admissible heuristic estimates, and in fact it may perform better with inadmissible, but more accurate heuristics. We show how Bugsy performs with admissible heuristics, and with two different types of corrected heuristics. Overall, we conclude that the best configuration is Bugsy with the standard admissible heuristics.

We discussed expansion delay in Section 5.1. We show results that demonstrate that using expansion delay is better than simply using d as the estimate of expansions to the goal. Then we compare two variants of Bugsy: one that ignores newly generated nodes that are found to already be on the closed list (we call these *duplicate* nodes) and one that reinserts these nodes onto the open list if they have better utility estimates than their previously closed version. Ignoring duplicates always performs better in some domains, and in others it performs better only when the preference is for very short search times.

Then, we compare A*, Speedy search, monitored ARA*, weighted A* with a learned weight, and Bugsy. We find that the simplest approach of learning a good weight for weighted A* gives the best performance. We also find that Bugsy, which doesn't use any off-line training, performs about as well as monitored ARA*, which does use off-line training. Therefore, if training instances are available, we recommend the simple weighted A* approach where the weight is selected based on performance on the training set. If no training instances are available Bugsy is the algorithm of choice.

Lastly, we compare Bugsy to both real-time search and DTA* on the platform pathfinding domain. In both experiments, Bugsy achieves the best utility.

6.2 Domains

In order to verify that our results hold for a variety of different problems, we performed our experiments on four different domains. The domains that we used are described briefly in the following paragraphs, with more detailed descriptions given in Appendix B.

6.2.1 15-Puzzle

The 15-puzzle is a popular heuristic search benchmark that has a small branching factor and few duplicates. For this domain, we used the reasonably informed Manhattan distance heuristic, and our implementation followed the heavily optimized solver presented by Burns, Hatem, Leighton, and Ruml (2012). We ran the 100 instances created by Korf (1985), and in plots including A* we only use results on the 94 instances solvable by A* in 6GB of memory.

6.2.2 Pancake Problem

The pancake problem is another standard puzzle with a large constant branching factor. In our experiments, we used instances with 50 pancakes, and the gap heuristic (Helmert, 2010). Since many of these problems were too difficult for A*, we used IDA* instead of A* on this domain.

6.2.3 Platform Pathfinding

The platform domain is a pathfinding domain of our own creation with dynamics based on a 2-dimensional platform-style video game, where the player must jump between platforms to traverse a maze. Video games often naturally have an element of time pressure. It has a large state-space and many cycles, but a reasonably informed heuristic based on visibility navigation. The instances used in our experiments were created randomly, using the generator described in Appendix B. This domain is also of particular interest because its action costs are given in units of time (each action is 50ms), so the objective of minimizing goal achievement time can be expressed as a linear combination of search time and solution cost.

6.2.4 Grid Pathfinding

Grid pathfinding is a popular heuristic search benchmark, motivated by robotics and video games. In our experiments, we used two different cost models, and two different movement models. The cost models were the standard *unit-cost* model and the *life-cost* model which assigns action costs such that the shortest, most direct path is more expensive than a longer, more circuitous path. This captures the popular adage that time is money. Instances were 5,000×5,000 grids with uniformly distributed obstacles. Our heuristics were based on the Manhattan distance heuristic for four-way grids, and the octile distance heuristic for eight-way grids. The octile distance heuristic is a simple modification to Manhattan distance that multiplies the shorter of the horizontal and vertical displacement by $\sqrt{2}$ to accounts for eight-way move costs.

6.3 Anytime Profile Accuracy

We want to ensure that our implementation works well and our training instance sets are representative enough that monitored ARA* can perform at its best. In this subsection, we evaluate the accuracy of the stopping policies created using the estimated anytime profiles by comparing them to an oracle. Since the stopping policy is only guaranteed to be optimal for the true algorithm profile, it is a matter of empirical study to determine whether or not the estimated profile will lead to a good policy.

To estimate the profile used by the monitored version of ARA*, we ran ARA* with a 6GB memory limit or until convergence on 1,000 separate test instances for each domain. Next, we created a histogram by discretizing the costs and times of each of the solutions into 10,000 bins (100 × 100). We experimented with different utility functions by varying the ratio w_f/w_t in Equation 1. Small values of w_f/w_t give a preference to finding solutions more quickly, whereas large values prefer finding cheaper solutions. In the case of the platform game, for example, this can be viewed as a way to change the speed at which the agent moves: a slow agent might benefit from more search in order to find a shorter path, but a fast agent can execute a path quickly, and may prefer to find any feasible solution as fast as possible.

Figure 3 shows the results of this experiment. The box plots represent the distribution of utility values found by ARA* using the estimated stopping policy, given as the factor of the oracle's utility. The oracle finds all solutions of the anytime algorithm until it converges on the optimal solution, then it picks the solution which would have maximized the utility function. Since the utility values are negative, larger factors represent smaller (more negative) utilities and thus a worse outcome. The boxes surround the second and third quartiles, the whiskers extend to the extremes, and circles show values that are more than 1.5× the inter-quartile range outside of the box. The center line of each box shows the median, and the gray rectangles show the 95% confidence interval on the means. Each box represents a different w_f/w_t as shown on the x axis. There is a reference line drawn across at $y = 1$ (the point where the oracle and estimated policy performed equally as well), and in many cases the boxes are so narrow that they are indistinguishable from this line.

Some points in these figures lie very slightly below the $y = 1$ line, indicating instances where the oracle performed worse than the estimated policy. This is possible due to the variance in solving times. In our experiment, the ARA* runs used to compute the oracle's

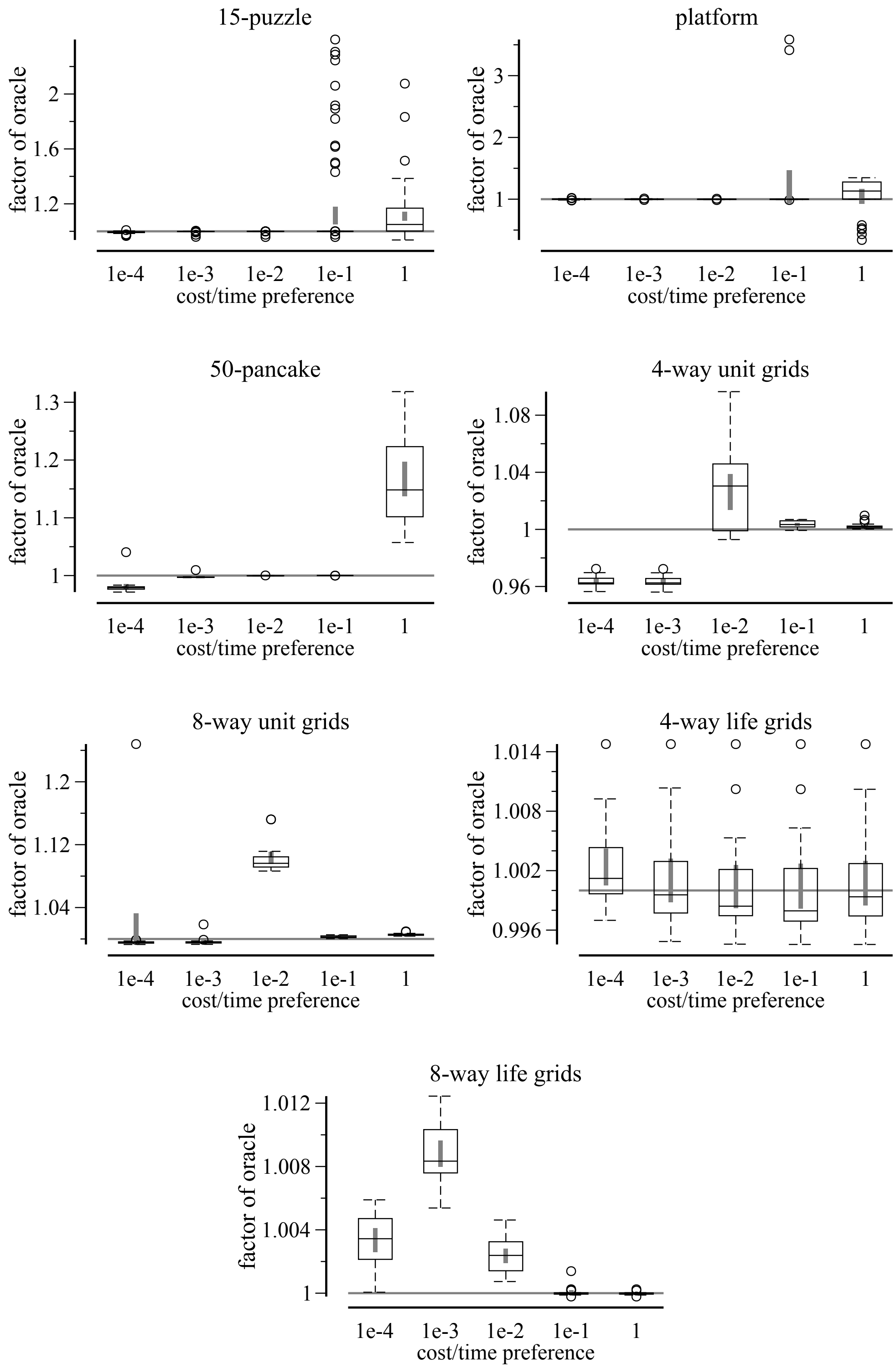

Figure 3: Comparison of the optimal stopping policy and the learned stopping policy.

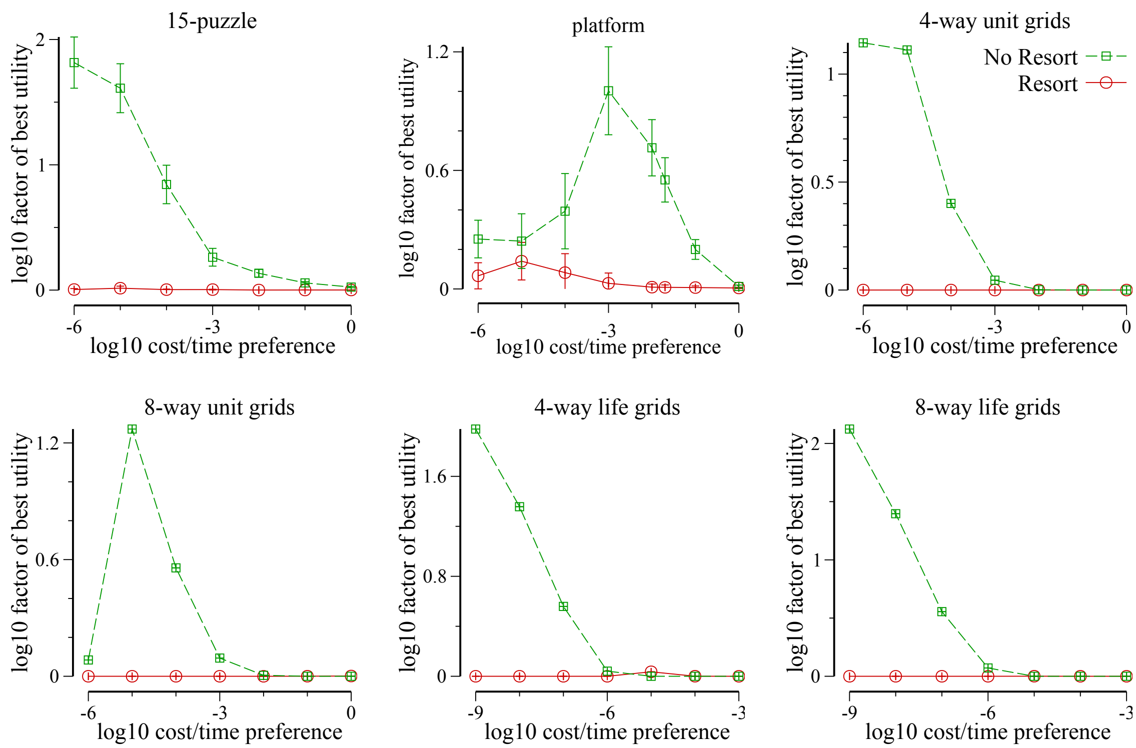

Figure 4: Bugsy: Resorting the open list (circles) vs not (boxes).

utilities occasionally found solutions more slowly than the ARA* runs using the estimated stopping policy. In other words, it is caused by the non-determinism inherent in a utility function that depends on solving time. As is obvious in the figure, these instances are quite rare and usually happened for small values of w_f/w_t, where miniscule time differences have a large effect on utility.

From these results, we conclude that our monitored ARA* implementation performs quite well, as the stopping policy often stopped on the best solution available from those emitted by the underlying anytime algorithm.

6.4 To Resort or Not to Resort?

In Section 5.5 we proved that re-sorting Bugsy's open list on power-of-two expansions only added a constant overhead per-expansion when amortized over the search. It is a matter of empirical evaluation to determine whether or not this overhead is worth the effort. While other re-sorting schedules are possible, we only tried re-sorting on power-of-two expansions.

Figure 4 shows the utility achieved by Bugsy both with and without re-sorting. The x axes show the w_f/w_t ratio determining the preference for solution cost and search time on a $\log_{10}$ scale. As with the previous plots, smaller values indicate a preference for faster search times and larger values indicate a preference for cheaper solutions. The y axes show the factor of the utility achieved by the best technique on each instance, again on a $\log_{10}$ scale. A y value of $\log_{10} 1 = 0$ indicates the best utility achieved by any technique on a given instance; values greater than zero indicate less utility. Points show the mean value over all test instances with error bars giving the 95% confidence intervals. From these plots, we can see that re-sorting the open list led to significant improvements in all domains. On the pancake puzzle, Bugsy without re-sorting was unable to solve any of the instances within a 6GB memory limit. In our remaining experiments, we always enable re-sorting on an exponential schedule.

6.5 Heuristic Corrections

In Section 5.4, we mentioned that Bugsy does not require admissible heuristic estimates, as it provides no guarantees on solution cost. In this section we compare Bugsy using the standard admissible heuristics to Bugsy using both on-line and off-line corrected heuristics. Following Thayer et al. (2011), our on-line heuristic correction used a global average of the single-step heuristic error between each node and its best offspring, and our off-line heuristic was a linear combination of h, g, $depth$, and d, for each node. The coefficients for each term in the off-line heuristic were learned by solving a set of training problems and using linear least squares regression.

The comparison is shown in Figure 5. The plots are in the same style as Figure 4. Typically the on-line correction technique performed worst—some times significantly worse—than the other two. We attribute this to its poor accuracy as observed by Thayer et al. (2011). On some problems, such as the 15-puzzle and the 8-way unit-cost grid pathfinding, the off-line correction technique performed best, but in general the simple admissible heuristics were the best or were competitive with the best. For the remainder of our experiments, we chose to use the simplest variant without any corrections as it did not require any off-line training (which is one of Bugsy's main benefits), and it was never the worst and was often the best or near the best.

6.6 Expansion Delay

In Section 5.1 we described why simply using d as an approximation for $exp(n)$, the number of nodes expanded to arrive at the goal beneath node n, is inaccurate. The search algorithm does not expand just the nodes along the path to a goal, but instead it vacillates between different solutions. To account for the search vacillation, we choose to estimate $exp(n) = delay \cdot d(n)$, where $delay$ is the average expansion delay—the average number of nodes expanded before the search makes progress along a single path to a goal. In this subsection, we show experimentally that using expansion delay provides much better performance than using d alone.

Figure 6 shows two versions of Bugsy: one that uses expansion delay, labelled "With Exp. Delay," and one that does not, labelled "Without Exp. Delay." It is clear from this figure that using expansion delay is beneficial. Also, we can see that on the right side of the plots, where cheaper solutions are preferred to short search times, using expansion delay is about the same as just using d by itself. This is because w_f is relatively large compared to w_t for these utility functions, so the $exp(n)$ term has little influence on the utility estimates.

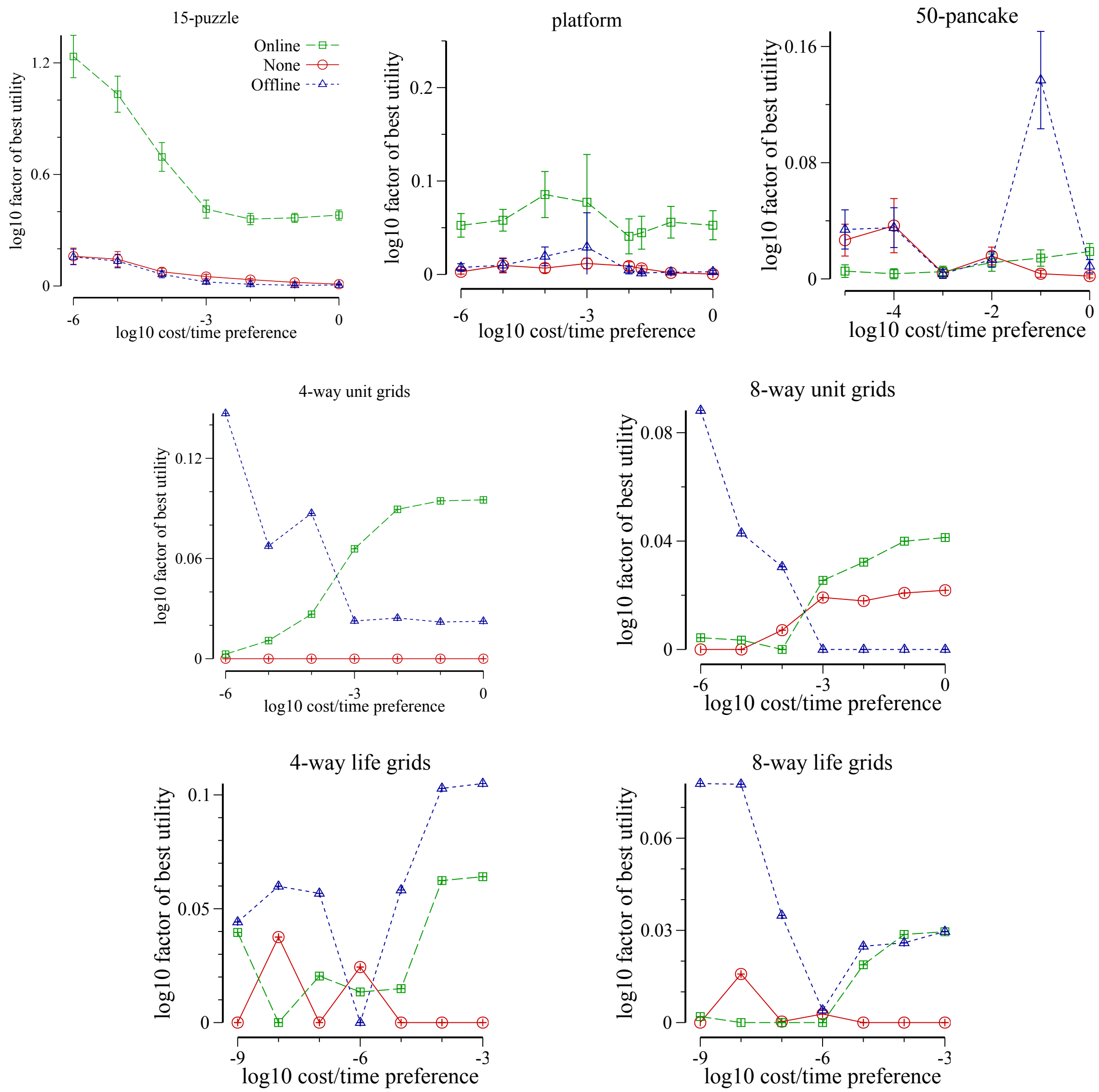

Figure 5: Bugsy: Heuristic corrections.

6.7 Duplicate Dropping

Suboptimal search algorithms do not expand nodes in strict order of increasing f. Consequently, they can expand a node, and later re-generate the same node via a cheaper path. We call such re-generations *duplicates*, and when they are generated via cheaper paths we say that they are *inconsistent*, because their current path cost (and subsequently the cost of the paths to all of their descendants) is more expensive than necessary (Likhachev et al., 2003). In the face of inconsistent nodes, a search algorithm can put the already expanded node back on the open list with a cost that accounts for the new, cheaper path. When the node comes to the front of the open list, it will be re-expanded and the inconsistency

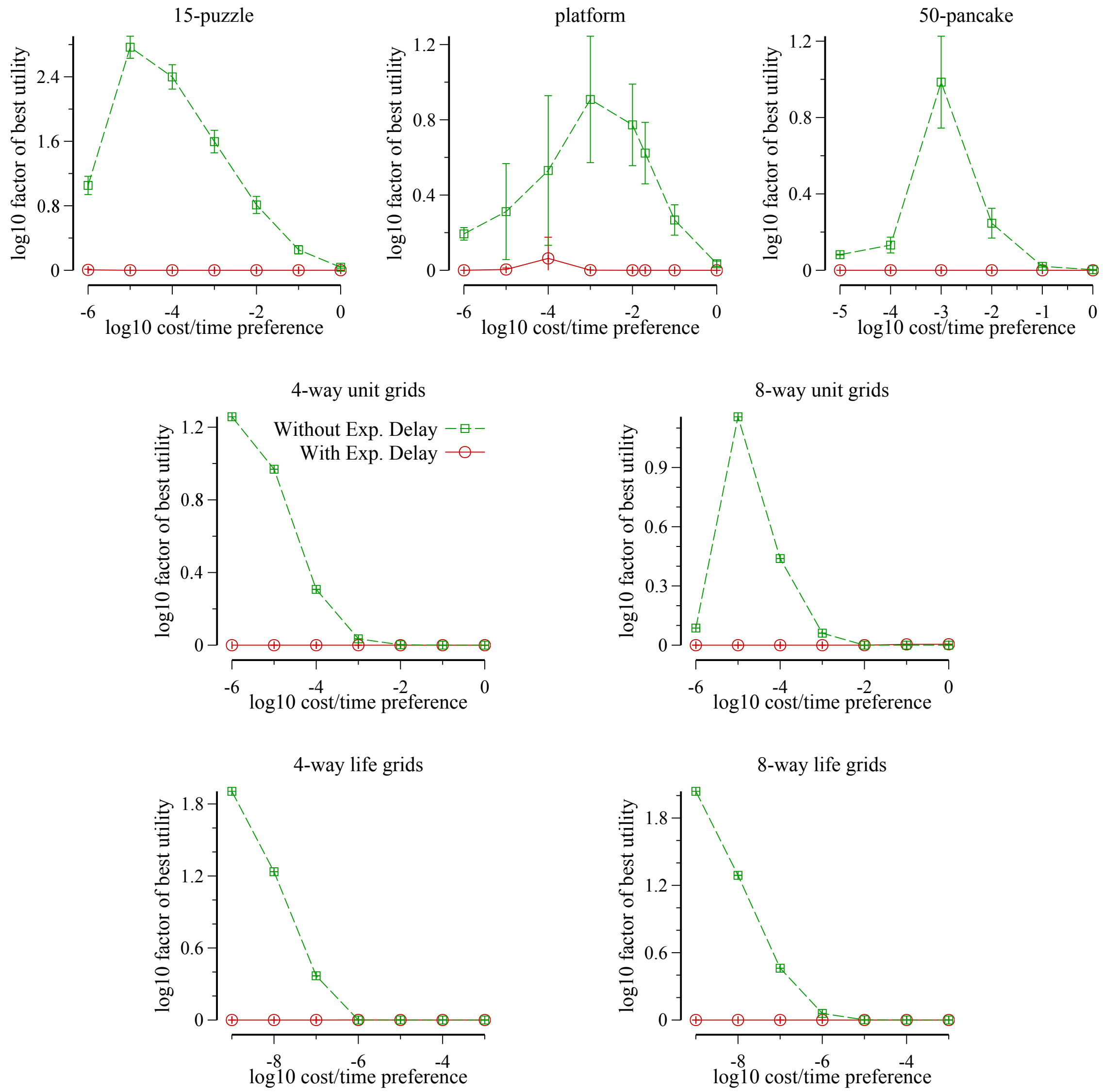

Figure 6: Bugsy: Expansion delay.

will propagate through to all of its descendants. Unfortunately, if there are a lot of such inconsistencies, then the search algorithm can spend a lot of time re-expanding the same nodes over and over again. An alternative technique is to simply ignore the inconsistency and drop all duplicate nodes when they are generated. Dropping duplicates can reduce the search effort needed to find a goal at the cost of finding more expensive solutions. Whether or not dropping duplicates is beneficial typically depends on the domain (Thayer & Ruml, 2008).

Figure 7 shows a comparison of Bugsy both with and without duplicate dropping. On the platform, tiles, and pancake domains using duplicate dropping is nearly always better than re-expanding duplicates. On the grid pathfinding problems,—with the notable excep-

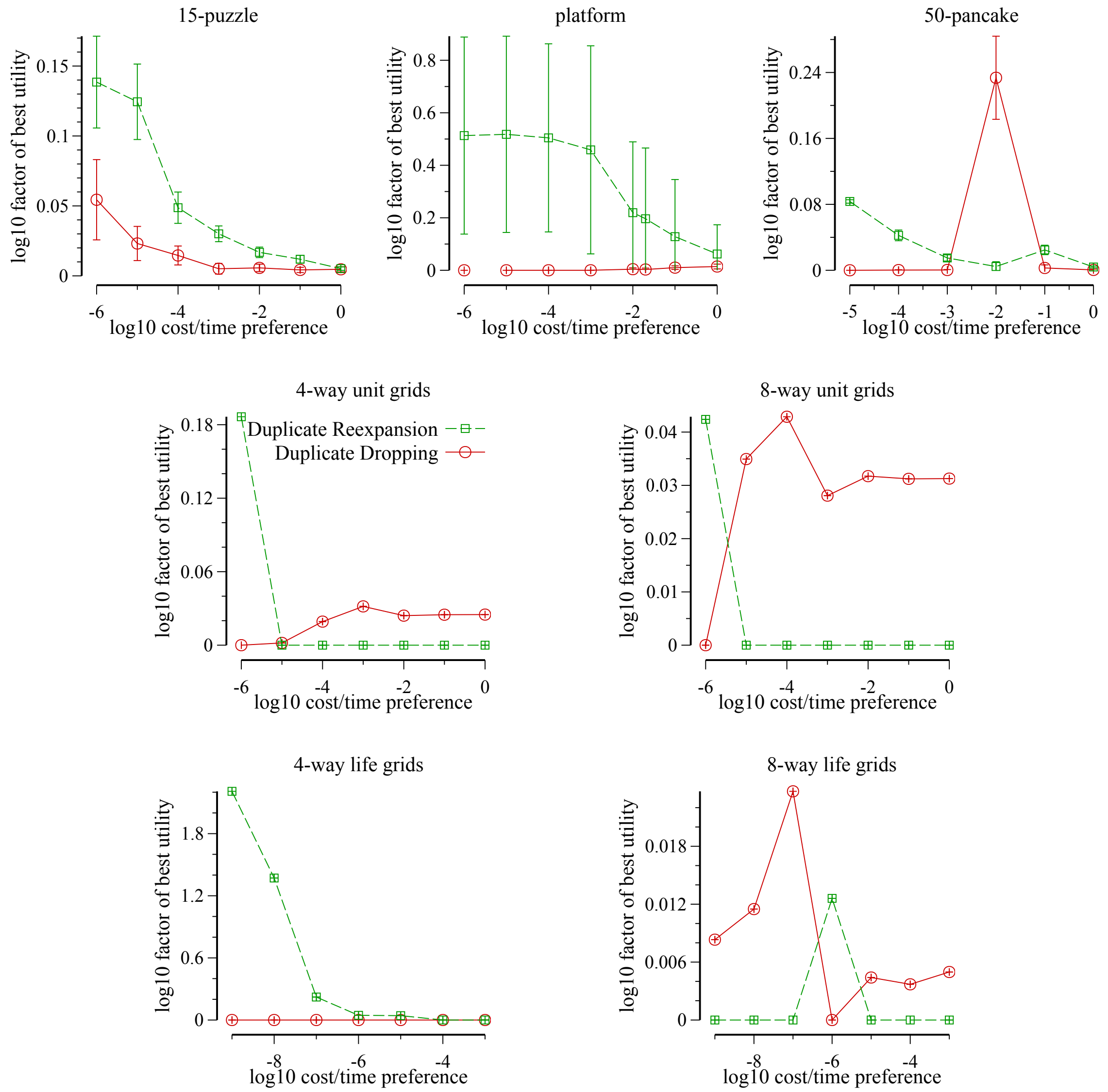

Figure 7: Bugsy: Duplicate dropping.

tion of 4-way life-cost grids—re-expanding duplicate nodes seems to give better performance except where solutions are needed as quickly as possible (on the left-hand side of the plots). This is reasonable, because duplicate dropping tends to sacrifice solution cost in order to reduce search time. Note also, that the values on the y axes of these plots are very small, so while the results are statistically significant, the difference between the two techniques on the grid problems where duplicate re-expansion performs better is quite small. In the next section we will see that A* actually achieves the most utility in many of the cases where duplicate re-expansion outperforms duplicate dropping.

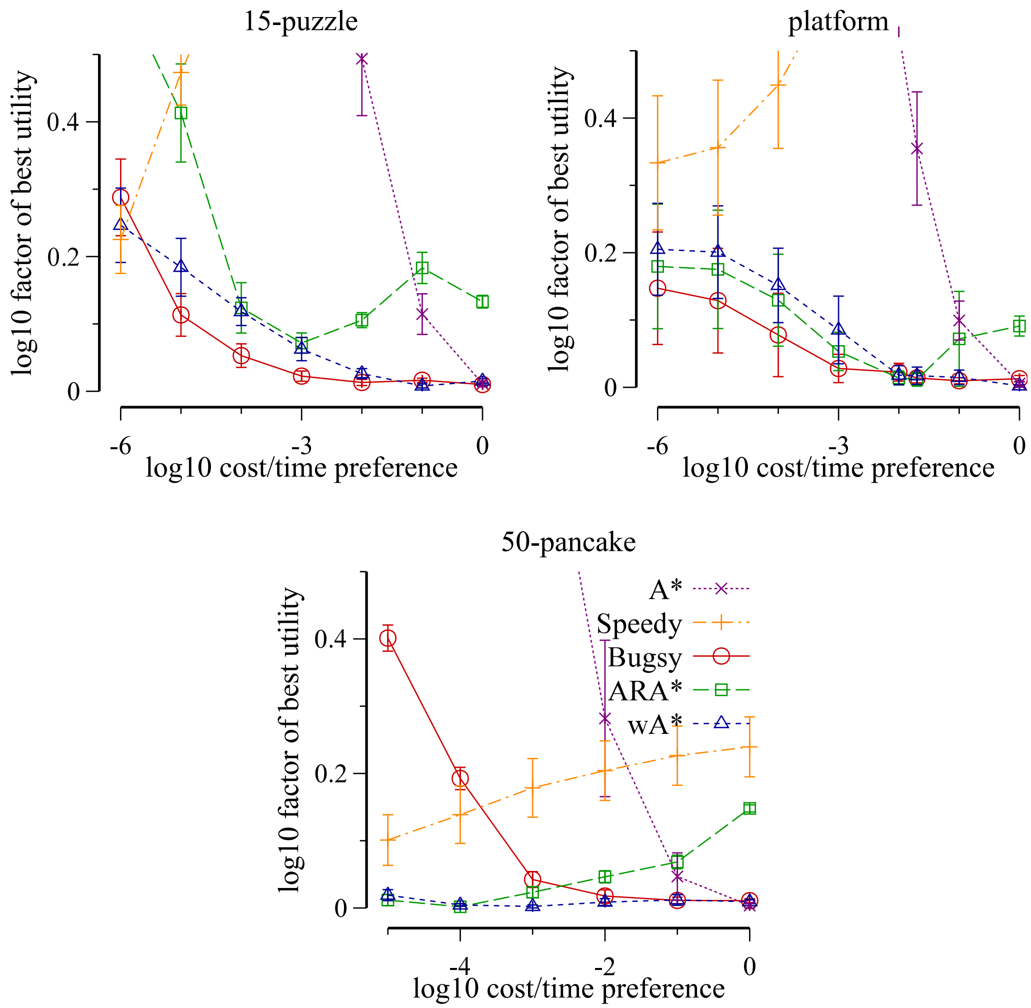

Figure 8: Comparison of techniques.

6.8 Comparing Techniques

Now that we understand the most promising configurations of the techniques we are studying, we can finally turn our attention to comparing them.

Figures 8 and 9 show a comparison of the three different techniques for utility-aware search. These plots are larger than the previous plots to improve clarity, because they have more lines. The plots include A*, Speedy Search, Bugsy, ARA* with monitoring (ARA*), and weighted A* with the weight chosen automatically for each different utility function from the set 1.1, 1.5, 2, 2.5, 3, 4, 6, and 10 (wA*). As we would expect, when the preference was for shorter search times (on the left-end of the x axis), A* performed poorly, as it stubbornly stuck to optimal solutions. Speedy search, however, performed quite well. As the preference shifted toward desiring cheaper solutions, A* began to do better whereas Speedy did worse. The utility-aware techniques were much more robust than both A* and

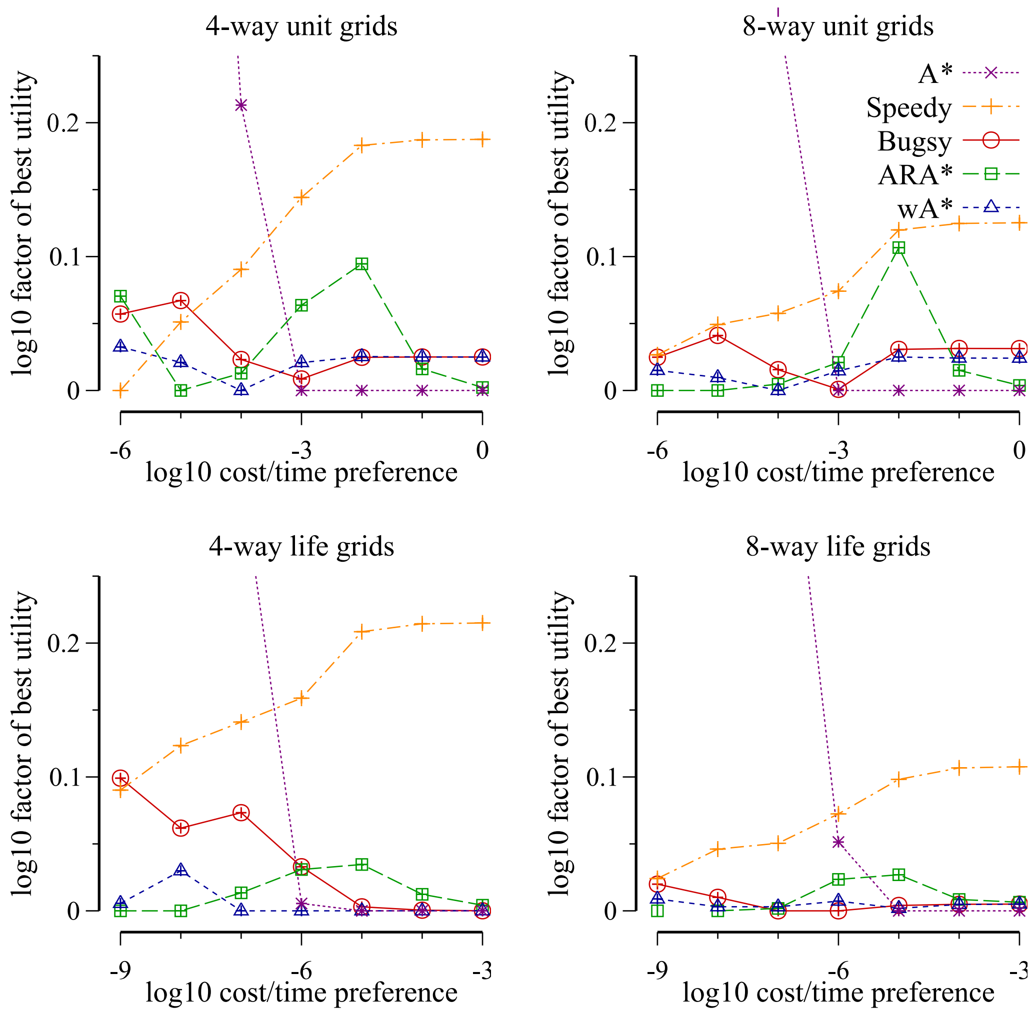

Figure 9: Comparison of techniques (continued).

Speedy, neither of which take the user's preference for search time and solution cost into account at all.

Of the utility-aware techniques, both Bugsy and weighted A* with an automatically selected weight performed the best. Bugsy was better on both the 15-puzzle and the platform domain. On the grid problems, Bugsy and weighted A* had roughly the same performance on the right side of the x axes. On the left side, Bugsy tended to get worse relative to the other utility-aware techniques, and ARA* with an anytime monitor was often the best performer. However, ARA* performed significantly worse in the middle and on the right-hand portion of the plot in some domains, leading us to recommend the weighted A* technique as a simpler and more robust approach.

The utility-aware techniques often performed as well as A* when low-cost solutions were preferred. When fast solutions were preferred, these techniques sometimes outperformed Speedy search. This likely indicates that solution cost still played a roll in the final utility

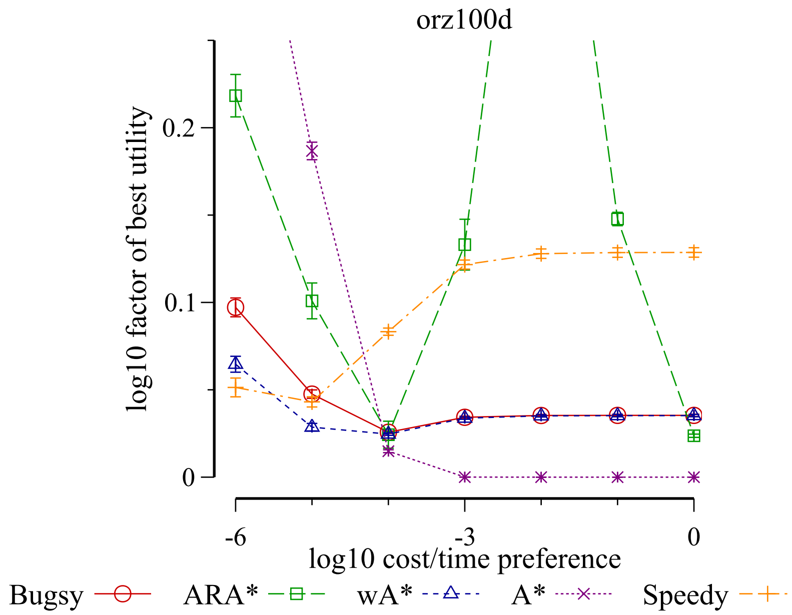

Figure 10: Grid pathfinding on a video game map.

on the left-most points in some of the plots. ARA* tended to achieve greater utility than Bugsy when solutions were needed quickly, but when cheaper solutions were preferred, Bugsy tended to be better than ARA*. On most domains, ARA* had a spike of low utility for ratios between 0.001, and 1, with the peak appearing between 10^{-6} and 10^{-3} for life-cost grids. This peak approximately coincides with the utility functions for which the estimated profile performed worse than the oracle as shown in Figure 3, possibly indicating that more than 1,000 training instances were required for these utility functions.

Overall, the utility-aware techniques were able to achieve much greater utility than the utility-oblivious A* and Speedy algorithms. This is not terribly surprising. Surprisingly, the results also suggest that our very simple parameter tuning technique can often give the best performance if a representative set of training instances is available. If not, then Bugsy is the algorithm of choice as it performs well and does not require any off-line training. Indeed, by putting reasoning about search time into the search algorithm itself, Bugsy can be competitive with techniques requiring previous experience.

6.9 Limitations

In the previous set of experiments, we saw that the utility-aware algorithms outperformed both Speedy search and A* for a wide range of utility functions. In this section, we look at one domain for which this tends not to be the case: video game grid maps.

Video games are one of the main motivations for research in grid pathfinding problems. Sturtevant (2012) observed that grid maps created by game designers often exhibit very different properties from maps generated algorithmically. Figure 10 shows a comparison of Bugsy, monitored ARA*, weighted A* with an automatically selected weight, Speedy, and

Figure 11: Grid pathfinding on a video game map.

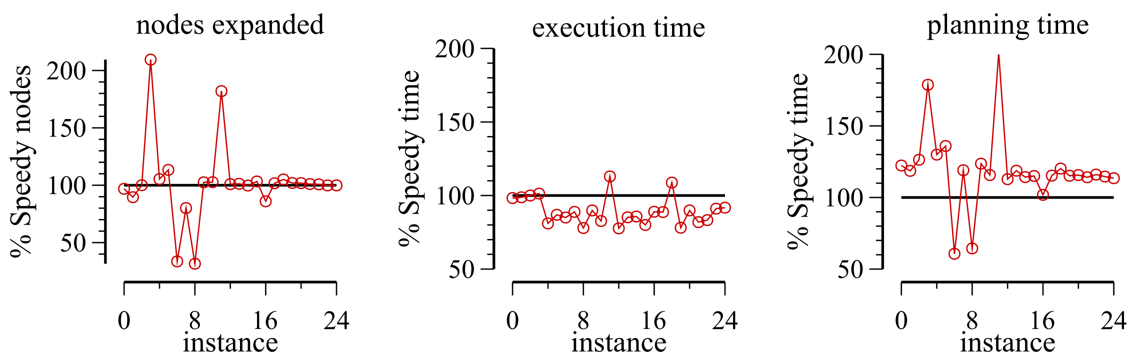

Figure 12: Nodes expanded, search time, and execution time.

A* on the Dragon Age Origins map orz100d from the benchmark set of Sturtevant. This map is shown in Figure 11. It has a fairly wide-open area at the top, with a more closed-off bottom half containing rooms and hallways. The format of the plot in this figure is the same as those in the previous subsection. As we can see, A* gave the best performance for a large range of utility functions, and Bugsy actually never outperformed Speedy or A* in the entire experiment (neither did ARA*, and wA* only gave the best performance at a single data point). We hypothesized that Bugsy's poor performance was because these problems are very easy to solve, and Bugsy's extra computation overhead, while very small, was more prominent.

To explore this hypothesis, we plotted the performance of Bugsy given as the difference from that of Speedy using a single utility function given by $w_f = 10^{-6}, w_t = 1$. This is the left-most utility function in Figure 10, a function for which here Speedy search performed the best and Bugsy performed poorly. Figure 12 shows the number of nodes expanded, the time spent executing, and the time spent searching for Bugsyas percentages of the equivalent values for Speedy search. The data points were gathered on a random sample of 25 instances from Sturtevant's (2012) scenario set for the orz100d map. Values below the line at 100% represent instances where Bugsy expanded fewer nodes or spent less time searching or executing, and values above the line represent instances where Bugsy expanded more nodes or spent more time than Speedy. The x axes shows the rank of the instances in the sample in increasing order of their optimal solution lengths.

As we can see in Figure 12, Bugsy expanded about the same number of nodes and had very similar execution times to Speedy. For problems with larger optimal solution costs Bugsy had slightly less execution time. The major difference in performance between these two algorithms, however, is shown in the right-most plot where we can see that Bugsy required more search time than Speedy search on almost every instance. Since Bugsy and Speedy expanded about the same number of nodes, this additional time must be due to Bugsy's small amount of extra overhead incurred from re-sorting and computing utility. We conclude that, barring this extra overhead, Bugsy would have performed as well as the best performer for this utility function. In domains where node expansion and heuristic computation isn't so simplistic, this overhead would be insignificant.

6.10 Training Set Homogeneity

In Section 6.8 we showed that our weighted A* approach outperformed other techniques in all domains, with the notable exception of the platform domain and 15-puzzle, where Bugsy was the best. Additionally, compared to other domains, the weighted A* technique performed relatively poorly on video game pathfinding (cf. Figure 10 where wA* is outperformed by the utility oblivious approaches at all points except for one). We believe that the poor performance of wA* on these domains is due to heterogeneous training sets. To verify this, we looked at the mean and standard deviation in the optimal path lengths for problems in all of our domains. The optimal path length can be viewed as a proxy for problem difficulty, and a high standard deviation in this statistic points to a diverse set of instances—some very easy to solve, and some quite difficult. For both the platform and video game path finding domains, the standard deviation in optimal path length was greater than 50% of the mean; more than twice that of the other domains. Note that, in domains such as the video game map, the variety in the layout of different areas of the map means that instances will inherently differ in their characteristics—merely gathering more instances will not produce a more homogeneous set. This evidence supports our hypothesis that weighted A*'s performance can be greatly hindered in situations where a representative training set is not available.

6.11 Real-time Search

The main focus for our study is algorithms for off-line search—they find entire paths to the goal before any execution begins. In real-time search (Korf, 1990), search and execution can happen in parallel, but an agent is only allowed a fixed amount of time to plan before it must perform each action. Real-time search has the possibility of being more efficient than off-line search in terms of goal achievement time, because if all search happens in parallel with execution, then goal achievement time is simply the execution time plus the small amount of time required to find the very first action. This in contrast to the off-line approach where goal achievement time is the sum of the entire search time and the execution time. In some situations, however, starting execution before having a complete plan to a goal is not acceptable, as it may lead an agent into a dead-end from which it can no longer reach any goal, so real-time search may not be applicable. Examples of domains with dead-ends include robotics, manufacturing (Ruml, Do, Zhou, & Fromherz, 2011), and spacecraft control: exactly those applications involving high value or danger, where automation is most worthwhile. In these cases, it is desirable to find an entire plan guaranteed to reach the goal, before any execution begins.

Hernández, Baier, Uras, and Koenig (2012) introduce a model for comparing real-time algorithms to off-line techniques such as A*, called the *game time model*. The game time model partitions time into uniform intervals, and the agent can execute a single action during each interval. Path planning can happen in parallel with execution (the agent can plan step t during the execution of step t-1), and the goal is to move the agent from its start location to its goal location in as few time intervals as possible, minimizing goal achievement time, the same objective that we discuss in Section 1. The game time model is a special case of the utility functions considered in this paper where solution cost is given in discrete, fixed-duration units of time.

Real-time search provides two benefits: first, it may be possible to reduce the goal achievement time by allowing search and execution to happen at the same time, and second, the agent can start moving toward its goal right away—a necessary property for video games. This leaves us with the question of whether or not real-time search algorithms can achieve better goal achievement time than the off-line utility-aware methods. On one hand, real-time search algorithms spend very little time searching without making progress toward the goal. On the other hand, real-time search algorithms tend to make decisions based on very local information and can find more costly solutions. In their results, Hernández et al. report that their best approach solves problems on initially-known grid maps in about the same number of time intervals as A*. In the previous section, we showed that the utility-aware techniques outperformed A* for most utility functions. In this section, we compare a state-of-the-art real-time search algorithm called LSS-LRTA* (Koenig & Sun, 2009) to BUGSY on the platform pathfinding domain.[4]

As with our previous experiments, we tested all algorithms with a variety of values for the ratio w_f/w_t. Since we are interested in goal achievement time, we set $w_t = 1$ and we calculate search time in units of seconds. This means that w_f represents the number of seconds in one unit of execution cost—the speed of the agent. We set the real-time constraint for LSS-LRTA* such that it was allowed to plan for the duration of one unit of execution, and so it always had its next action ready for execution when its currently executing action completed.

4. We do not compare to Time-bounded A* (TBA, Björnsson, Bulitko, & Sturtevant, 2009), the method that performed the best for Hernández et al. (2012), because the platform domain forms a directed search graph, and TBA* only works on undirected search graphs. We also did not compare with the newer f-LRTA* (Sturtevant, 2011), because it did not perform as well as LSS-LRTA* on the platform domain, which has directed edges.

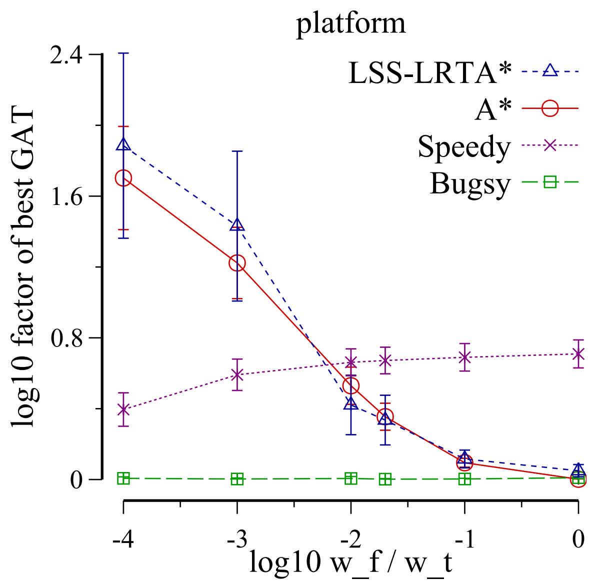

Figure 13: Comparison of Bugsy and real-time search.

Figure 13 shows the results of the comparison. As we can see, LSS-LRTA* gives rather poor performance; its goal achievement times nearly match that of A*, but Bugsy was able to achieve the goal much faster. This shows that simply allowing search and execution to take place in parallel is not sufficient to reduce goal achievement time; it can be better to spend time searching the solution all of the way to the goal if the alternative is to spend a long time executing a poor plan.

6.12 Decision-theoretic A*

Decision-theoretic A* (DTA*, Russell & EricWefald, 1991) is a utility-aware algorithm that allows for concurrent search and execution. It is based on ideas from real-time heuristic search, but unlike traditional real-time search, where each action is emitted after a fixed amount of search, DTA* decides when to stop searching and emit an action using a decision-theoretic analysis. At any time there is a single best top-level action with the lowest cost estimate. The search emits this action if it is decided that the utility of emitting the action outweighs the utility of further search. DTA* uses an approximation (found by off-line training) of how the solution cost estimate for each top-level action improves with additional search. Using a consistent heuristic, this estimate can only increase (Nilsson, 1980), so DTA* stops searching when it decides that the time required to raise the best action's estimated cost to the point that it is no longer the best action will be more costly than the expected gain from determining that there is a different best action.

Compared to Bugsy, DTA* is relatively myopic because it only considers the cost of search involved in selecting individual actions. DTA* does not consider the additional search required by the solution path to which it commits by choosing an action. Where Bugsy uses both d and expansion delay to reason about the required search effort for the entire

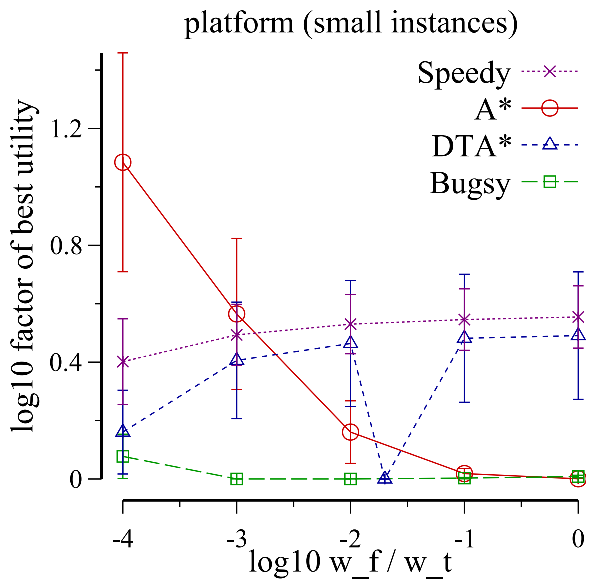

Figure 14: Comparison of Bugsy and DTA*.

path beneath a node, DTA* only reasons about the search required to determine the best action to emit right now.

We implemented DTA* to assess how a utility-aware real-time search might compare to a utility-aware off-line search for planning under time pressure. Figure 14 shows the results of a comparison between DTA* and Bugsy on the platform pathfinding domain. Unfortunately, DTA* had fairly poor performance, so our experiment used smaller instances consisting of 25x25 blocks, instead of the 50x50 block instances used in previous experiments. Following Russell and EricWefald (1991), we gathered off-line training data for DTA* using states sampled uniformly with a probability of 0.1 from among those visited by a real-time search algorithm. Russell and EricWefald (1991), used an algorithm called SLRTA*, but we used LSS-LRTA*, because it is the current state of the art. Our training set consisted of 100 25x25 platform instances. We also verified our implementation by ensuring that it compared favorably to A* and Speedy search on the 15-puzzle—the same domain used by Russell and EricWefald—using a variety of different utility functions. From Figure 14, we can see DTA* often had significantly worse utility than Bugsy, often performing only slightly better than Speedy search, and sometimes performing worse than A*, for example, when cheap solutions were desired.

7. Related Work

Because Bugsy uses estimates of its own search time to select whether to terminate or continue, and to select which node to expand, it may be said to be engaging in metareasoning, that is, reasoning about which reasoning action to take. There has been much work on this topic in AI since the late 1980s (Dean & Boddy, 1988) and continuing today (Cox & Raja, 2011).

Dean and Boddy (1988) consider the problem faced by an agent that is trying to respond to predicted events while under time constraints. Unlike our setting, their concern is with choosing how much time to allocate for prediction and how much to allocate for deliberation. To solve this type of time-dependent planning problem, they suggest the use of (and also coined the term) *anytime algorithms*. Unlike the anytime-based techniques discussed previously, which attempt to find a stopping policy to optimize a utility function, Dean and Boddy used anytime algorithms as a means for allowing different allocations of time between predicting and deliberation. Later, Boddy and Dean (1989) show how anytime algorithms and their time-dependent planning framework can be used by a delivery agent that must traverse a set of waypoints on a grid, by allocating time between the ordering of waypoints and the planning used to travel between them. Dean, Kaelbling, Kirman, and Nicholson (1993) also adapt the technique for scheduling deliberation and execution when planning in the face of uncertainty.

Garvey and Lesser (1993) present *design-to-time* methods that advocate using all available time to find the best possible solution. Unlike anytime approaches that can be interrupted at any time, the design-to-time method requires the time deadline to be given upfront. This way, the algorithm can spend all of its time focusing on finding a single good solution, instead of possibly wasting time finding intermediate results. Design-to-time also differs from *contract* techniques like DAS (Dionne et al., 2011), because in the design-to-time framework there must be a predefined set of solvers with known (or predictable) solution times and costs. The design-to-time method will select an appropriate solver for the problem and deadline, possibly interleaving different solvers if deemed appropriate. The information about cost and solutions times, which design-to-time methods require, is usually unavailable and must be learned off-line. Techniques like DAS and Bugsy, on the other hand, only use information that can be computed on-line.

Hansen, Zilberstein, and Danilchenko (1997) show how heuristic search with inadmissible heuristics can be used to make anytime heuristic search algorithms. Like the techniques presented in this paper, they consider the problem of trading-off search effort for solution quality. To this end, they propose one possible optimization function for anytime heuristic search search that attempts to maximize the rate at which the algorithm decreases solution cost. Like the anytime monitoring technique shown in Section 3.1, their evaluation function relies on learning the profile of the anytime algorithm offline. In their analysis of the 8-puzzle, they conclude that, while their method had good anytime behavior, there was little benefit of using it instead of trial-and-error-based hand tuning. This is not surprising given the strong performance demonstrated by offline-tuned weighted A* in our experiments.

More recently, Thayer et al. (2012) proposed an approach for minimizing the time between solutions in anytime algorithms. They demonstrate that their new state-of-the-art algorithm performs well on a wide variety of domains, and it can be more robust than previous approaches. Like Bugsy, their technique relies on using d heuristics to estimate the search effort required to find solutions. However, they only focus on solutions that will require the least amount of effort, and do not optimize for a trade-off of search time for solution cost.

In addition to controlling expansion decisions, metareasoning can also be used during heuristic evaluation. Often search algorithms will use the maximum value computed over multiple heuristics as a more accurate estimate of cost to goal. For some problems, like domain-independent planning, heuristics are quite expensive, so the increased accuracy gained via maximizing over many heuristics may not be worth the increased computation time. Domshlak, Karpas, and Markovitch (2010) introduce an on-line learning technique to decide on a single heuristic to compute for each state, instead of computing many and taking the max.

Other related work using metareasoning to control combinatorial search has been done in the area of constraint satisfaction problems (CSPs), and boolean satisfiability (SAT). Tolpin and Shimony (2011) use rational metareasoning to decide when to compute value ordering heuristics in a CSP solver. The focus of the work was on value ordering heuristics that gave solution count estimates; the solver only bothered to compute the heuristic at decision points where it was deemed worthwhile. Their experiments demonstrate that the new metareasoning variant outperformed both the variant that always computed the heuristic and one that computed the heuristic randomly. Horvitz, Ruan, Gomes, Kautz, Selman, and Chickering (2001) apply Bayesian structure learning to CSPS and SAT problems. They consider the problem of quasi-group completion, and unlike Tolpin and Shimony (2011) who use on-line metareasoning to control search, they use off-line Bayesian learning over a set of hand-selected variables to predict whether instances will be long or short running.

There has been a lot of work on attempting to estimate the size of search trees off-line (Burns & Ruml, 2013; Knuth, 1975; Chen, 1992; Kilby, Slaney, Thiébaux, & Walsh, 2006; Korf, Reid, & Edelkamp, 2001; Zohavi, Felner, Burch, & Holte, 2010). This is a related topic, as it is concerned with estimating search effort before an entire search has been performed. One may imagine leveraging such a technique to predict search time in an algorithm like Bugsy. Unfortunately, these estimation methods can be rather costly in terms of computation time, so they are not suitable as an estimator that is needed at every single node generation. Another possibility is to use off-line estimations to find parameters that affect the performance of search on a given domain. This knowledge could be helpful for creating the representative training sets used by algorithms like weighted A* and anytime monitoring, which require off-line training.

8. Conclusions

We have investigated utility-aware search algorithms that take into account a user-specified preference trading-off search time and solution cost. We presented three different techniques for addressing this problem. The first method was based on previous work in the area of learning stopping policies for anytime algorithms. To the best of our knowledge, we are the first to demonstrate these techniques in the area of heuristic search. The second method was a novel use of algorithm selection for bounded-suboptimal search that chooses the correct weight to use for weighted A* for a given utility function. The last technique that we presented was the Bugsy algorithm. Bugsy is the only technique of the three that does not require off-line training.

We performed an empirical study of these techniques in the context of heuristic search, investigated the effect of the parameters of each algorithm on performance, and compared the different techniques to each other. Surprisingly, the simplest technique of learning a weight for weighted A* was able to achieve the greatest utility on many problems, outperforming the conventional anytime monitoring approach. Also surprisingly, Bugsy, an algorithm that does not use any off-line training, performed just as well as the off-line techniques that had the advantage of learning from thousands of off-line training instances. If a representative set of training instances is not available then Bugsy is the algorithm of choice. Overall, the utility-aware methods outperformed both A* and Speedy search for a wide range of utility functions. This demonstrates that heuristic search is no longer restricted to solely optimizing solution cost, freeing a user from the choice of either slow search times or expensive solutions.

Unlike previous methods for trading deliberation time for solution quality, Bugsy considers the trade-off directly in the search algorithm—deciding, for each node, whether the result of expansion is worth the time. This new approach provides an alternative to anytime algorithms. Instead of returning a stream of solutions and relying on an external process to decide when additional search effort is no longer justified, the search process itself makes such judgments based on the node evaluations available to it. Our empirical results demonstrate that Bugsy provides a simple and effective way to solve shortest-path problems when computation time matters. We would suggest that search procedures are usefully thought of not as black boxes to be controlled by an external termination policy but as complete intelligent agents, informed of the user's goals and acting rationally on the basis of the information they collect so as to directly maximize the user's utility.

Acknowledgments

We greatly appreciate feedback and suggestions from Shlomo Zilberstein and Scott Kiesel. We would also like to think Richard Korf for pointing out the work of Shekhar and Dutta (1989). We are also grateful for support from the NSF (grant 0812141 and grant 1150068), the DARPA CSSG program (grant D11AP00242), and a University of New Hampshire Dissertation Year Fellowship. A preliminary version of Bugsy was presented by Ruml and Do (2007); see Appendix A. Elisabeth Crawford assisted with the original version during a summer internship at PARC.

Appendix A. Previous Bugsy

A previous version of Bugsy was proposed by Ruml and Do (2007), however, this early realization differs substantially from the one presented here. It used aggressive duplicate re-expansion, heuristic corrections, and it only used d to estimate the remaining expansions until a goal is reached. In Section 6.7 we showed that duplicate dropping outperforms duplicate re-expansion in many domains. We found that inadmissible heuristics performed poorly (cf Section 6.5) in practice, even when compared to the standard admissible estimates. Also, to temper the inadmissible of its corrected estimates, the previous Bugsy multiplied its heuristic estimates by an arbitrary weight $(\min(200, (w_t/w_f))/1000)$; our version does not require this ad hoc fix. We discussed why d is a poor estimate for the number of remaining expansions in Section 5.1, and in Section 6.6 we showed, experimentally, that using expansion delay performs much better than just using d alone.

Recall that Bugsy uses f and d to approximate the cost and path length of the best utility outcome that is enabled by the expansion of the node. Note, however, that the f and d function used throughout this paper refer to the cheapest solution beneath a node n, and this may not be the goal that results in the maximum utility. To better assess the available outcomes, the previous version of Bugsy computed two utility estimates for each node: one for the cheapest solution beneath the node and the other for the *nearest* solution in terms of node expansions. In non-unit-cost domains, these two estimates may differ. For example, on the life-cost grid pathfinding domains, the cheapest solution usually involves moving toward the top of the grid where actions are cheap, but the nearest solution will follow a straight-line path to the goal. In general, there can be a large number of different solutions through each search node, and the solutions may cover a whole spectrum of different cost/time trade-offs. By considering more than just the cheapest solution, as was done in our implementation, it may be possible to find solutions with better utility. On the other hand, it may be too costly to compute multiple heuristics for each node, so whether or not this modification is beneficial depends on the domain.

Appendix B. Domains

We performed our experiments on a variety of different domains, which we describe in further detail here.

B.1 15-Puzzle

The 15-puzzle is one of the most popular benchmark domains for heuristic search algorithms. It consists of a 4-by-4 frame into which 15 tiles have been placed. One slot of the board does not contain a tile, it is called the *blank*. Tiles that are above, below, left of or right of the blank may be slid into the blank slot. The objective of the 15-puzzle is to slide tiles around in order to transform an initially scrambled puzzle into the goal state with the blank in the upper-left corner and the tiles ordered 1–15 going from left to right, top to bottom. This domain is interesting because plans are hard to find, the branching factor is small and varies little from its mean of about 2.13 (Korf et al., 2001), there are few duplicates, and the heuristic is reasonably informed.

In our experiments we use the popular 100 15-puzzle instances created by Korf (1985). In plots that include A*, however, we only used the 94 instances solvable by A* in 6GB of memory. The average optimal solution length for these instances was 52.4. For our training set, we generated 1,000 instances using a 1 million step random walk back from the goal position. We used the Manhattan distance heuristic, which sums the vertical and horizontal distance that each tile must move to arrive at its goal position. Our implementation follows the heavily optimized solver presented by Burns et al. (2012).

B.2 Pancake Puzzle

The pancake puzzle (Dweighter, 1975; Gates & Papadimitriou, 1979) is another permutation puzzle. It consists of a stack of differently sized pancakes numbered 1–N. The pancakes must be presented at a fancy breakfast, so a chef needs to sort the originally unordered stack of pancakes by continually sticking a spatula into the stack and reversing the order of the pancakes above. Said another way, the pancake problem involves sorting a sequence of numbers by using only prefix reversal operations. This simple problem is interesting because it creates a search graph with a large branching factor (the number of pancakes minus one).

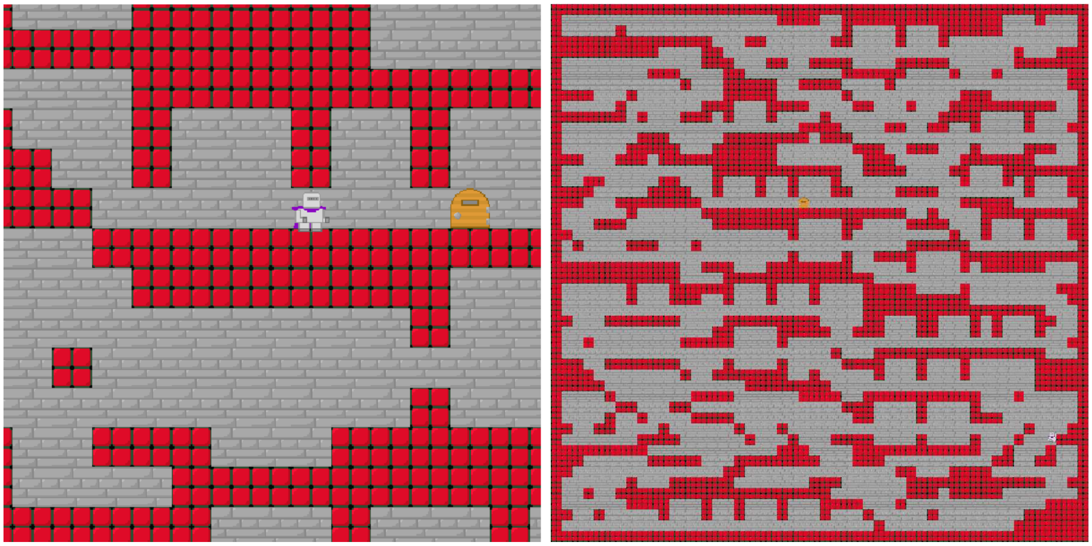

Figure 15: A screenshot of the platform pathfinding domain (left), and a zoomed-out image of a single instance (right). The knight must find a path from its starting location, through a maze, to the door (on the right-side in the left image, and just above the center in the right image).

For our experiments, we used 25 randomly generated 50-pancake puzzle instances, and our training set consisted of 1,000 randomly generated instances. We used the powerful GAP heuristic of Helmert (2010), which sums the number of pairs of adjacent pancakes that are not in sequence.

B.3 Platform Pathfinding

The platform domain is a pathfinding domain of our own creation with dynamics based on a 2-dimensional platform-style video game, written partially by the first author, called `mid`[5]. The left image of Figure 15 shows a screenshot from `mid`. The goal is for the knight to traverse a maze from its initial location, jumping from platform to platform, until it reaches the door. `Mid` is an open source game available from `http://code.google.com/p/mid-game`. For our experiments the game physics of the game were ported from C to C++ and were embedded in our C++ search codebase. We generated 1,000 training instances and 100 test instances using the level generator from `mid`. An example instance is shown on the right panel in Figure 15. The domain is unit-cost and has a large state space with a well-informed heuristic.

The available actions are different combinations of controller keys that may be pressed during a single iteration of the game's main loop: left, right, and jump. Left and right move to the knight in the respective directions (holding both at the same time is never considered by the search domain, as the movements would cancel each other out, leaving the knight in

5. The other author was Steve McCoy, who also drew the tile graphics shown in Figure 15.

place), and the jump button makes the knight jump, if applicable. The knight can jump to different heights by holding the jump button across multiple actions in a row up to a maximum of 8. The actions are unit cost, so the cost of an entire solution is the number of game loop iterations, called frames, required to execute the path. Each frame corresponds to 50ms of game play.

Each state in the state space contains the x, y position of the knight using double-precision floating point values, the velocity in the y direction (x velocity is not stored as it's determined solely by the left and right actions), the number of remaining actions for which pressing the jump button will add additional height to a jump, and a boolean stating whether or not the knight is currently falling. The knight moves at a speed of 3.25 units per frame in the horizontal direction, it jumps at a speed of 7 units per frame, and to simulate gravity while falling, 0.5 units per frame are added to the knight's downward velocity up to a maximum of 12 units per frame.

For further details on the platform domain, please refer to the source code repository given at the start of Section 6.

B.3.1 Level Generator

The instances used in our experiments were created using the level generator from `mid`, a special maze generator that builds 2-dimensional platform mazes on a 50×50 grid of blocks. Each block is either open or occluded, and to ensure solvability given the constraints imposed by limited jump height, the generator builds the maze by stitching together pieces from a hand-created portfolio. Each piece consists of a number of blocks that are either free or occluded, and a start and end location for which traversability is ensured within the piece. A piece can be added to the grid at any location for which it fits. A piece fits if it does not occlude a block that belongs to a previously placed piece. The maze is built using a depth-first procedure: a piece is selected at random and if it fits in the grid with its start location lined up with the end location of its predecessor then it is placed and the procedure recurs. The number of successors of each node is chosen uniformly from the range 3–9 inclusive, and the procedure backtracks when there are no pieces that fit on the previous block. Once the maze is constructed, blocks that do not belong to any piece are marked as occluded. The right image in Figure 15 shows a sample of a level generated by this procedure. The source code for the level generator is available in the `mid` source repository mentioned above.

B.3.2 Heuristic

We developed a heuristic for the platform domain that is based on visibility navigation (Nilsson, 1969). Each maze is pre-processed to convert its grid representation into a set of polygons representing each connected component of occluded cells in the level. The space is then scaled to account for the movement speed of the knight; the knight can fall faster than it can move in the horizontal direction, so the polygons end up squished vertically and stretched horizontally. The visibility navigation problem is then solved in reverse from the four corners of the goal cell to the center of every non-occluded cell of the maze. To maintain admissibility, the cost of each edge in the visibility problem is not the length of the visibility line, but instead is the maximum of the length of the line divided by $\sqrt{2}$ and the X and Y displacements between the end points of the line. This accounts for the fact

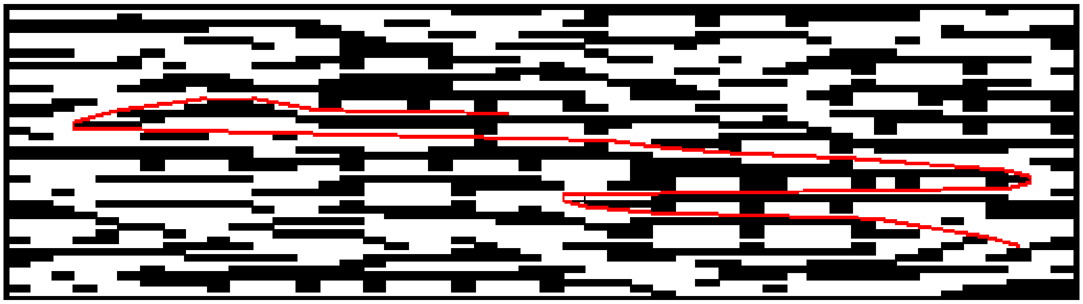

Figure 16: The visibility navigation instance for the platform domain's heuristic. The visibility path between the initial state and the goal state is drawn in red.

that the knight can be moving both horizontally and vertically at the same time, and that moving a distance of $\sqrt{2}$ in the scaled space still takes only a single frame.

During search, the heuristic value of a state is computed in one of two different ways. If the straight-line path from the center of the knight to the goal is not occluded then the maximum of the X and Y distances to the goal scaled down by travel speed is used as the heuristic estimate. Otherwise, the heuristic is the cost of the path in the visibility graph from the center of the cell that contains the knight's center point minus the maximum of the X and Y distance (in number of frames) of the knight's center point to the center of its cell. Figure 16 shows the same map from the right image of Figure 15, scaled, broken into polygon components, and with the visibility path between the initial state and the goal state drawn in red.

B.4 Grid Pathfinding

Our final domain was grid pathfinding. This is a very popular domain in both video games and robotics, as such it has garnered much attention in the heuristic search community. In our experiments, we used 5,000x5,000 grids with both four-way and eight-way connectivity and uniform obstacle distributions. For four-way connected grids, each cell was blocked with a probability of 0.35, and for eight-way connected grids cells were blocked with a probability 0.45. We also consider two different cost models, the standard *unit cost* model in which horizontal and vertical moves cost 1 and diagonal moves cost $\sqrt{2}$. The other is called the *life cost* model, where each move has a cost equal to the row number from which the move took place, causing cells toward the top of the grid to be preferred. With the life cost model, short direct solutions can be found quickly, however they will be relatively expensive, while a least-cost solution involves many annoying economizing steps (Ruml & Do, 2007). This model can be viewed as an instantiation of the popular belief that 'time is money,' as one can choose to incur additional cost for a shorter and simpler path. For each combination of movement model and cost model, we generated 25 test instances and 1,000 training instances. Finally, we used the Manhattan distance heuristic for four-connected grids and the octile distance heuristic for eight-connected grids. For the life cost model our heuristics also took into account the fact that moving toward the top of the grid then back down may be cheaper than a direct path.

Appendix C. Anytime Policy Estimation

It can be challenging to write algorithms that rely on off-line training data. If the algorithm behaves unexpectedly, then it is unclear if there is a bug in the implementation, a bug in the off-line learning procedure, or if the training set is merely insufficiently representative. In this appendix, we describe how we implemented and verified our procedure for estimating an anytime profile.

Figure 17 shows the pseudocode for building a profile based on the description given by Hansen and Zilberstein (2001). The algorithm accepts a set of solution streams as input, one stream for each solved instance, then proceeds in two steps. The first step is the Count-Solutions function that counts the number of times each solution cost was improved upon. The function iterates through each solution (line 5), computes the bin of a histogram into which its cost value falls (line 7), then for each subsequent solution a count is added to *qqtcounts* for each time step for which the first solution improved to the second solution. In addition, the number of total improvements for each solution and time bin are counted in the *qtcounts* array. The costbin and timebin functions bin cost and time values respectively by returning an integer for the corresponding index in the histogram:

$$\text{costbin}(q) = \left\lfloor \frac{q - q_{min}}{(q_{max} - q_{min})/ncost} \right\rfloor$$

$$\text{timebin}(\Delta t) = \left\lfloor \frac{\Delta t - \Delta t_{min}}{(\Delta t_{max} - \Delta t_{min})/ntime} \right\rfloor$$

.

The second step is the Probabilities function that converts the counts computed in the first step into normalized probability values. This is achieved by dividing the number of Δt steps for which a solution of cost q_i improved to a solution of cost q_j (*qqtcounts*) by the total number of steps for which a solution of cost q_i was improved (*qtcounts*, and lines 28). The probability values are "smoothed" by adding half of the smallest probability to each bin representing a solution cost improvement. This step removes zero-probabilities, allowing improvement to be considered. Finally, the probabilities are normalized so that the probability of all non-decreasing-cost solutions for each current cost and time step sum to one (lines 31–37). Once the profile is computed, it is saved to disk for later use when computing the stopping policy.

We found that it was extremely useful to have a simple way to validate our policies while debugging our implementation. One option is to create a stopping policy, run ARA* with monitoring on a handful of instances and with a handful of utility functions to verify that it gives the expected behavior. Unfortunately, this approach was rather cumbersome and prone to error, as it only evaluated the policy on the small number of instances that we were willing to run by hand. Instead, we chose to validate our implementation by plotting the polices generated from the training data on different utility functions. By plotting the extreme policies that only care about solution cost and search time, along with some intermediate policies that trade-off the two, it was much simpler to debug our code.

```
Profile(streams)
  qtcounts, qqtcounts ← Count-Solutions(streams)
  return Probabilities(qtcounts, qqtcounts)

Count-Solutions(streams)
  qtcounts ← new int[ncost][ntime] // Initialized to zero.
  qqtcounts ← new int[ncost][ncost][ntime] // Initialized to zero.
  for s in streams
      for i from 1 to |s|
          q_i ← costbin(s[i].cost)
          q_cur ← q_i, Δt_cur ← 0
          // Count cost at each time increment after solution i.
          for j from i + 1 to |s|
              q_next ← costbin(s[j].cost)
              Δt_next ← timebin(s[j].time − s[i].time)
              // Current solution cost up to the time of solution j.
              for Δt = Δt_cur to Δt_next − 1
                  increment qtcounts[q_i][Δt]
                  increment qqtcounts[q_cur][q_i][Δt]
              q_cur ← q_next, Δt_cur ← Δt_next
          // Last solution cost up to the final time.
          for Δt = Δt_cur to ntime
              increment qtcounts[q_i][Δt]
              increment qqtcounts[q_cur][q_i][Δt]
  return qtcounts, qqtcounts

Probabilities(qtcounts, qqtcounts)
  probs ← new float[ncost][ncost][ntime]
  for q_i from 1 to ncost
      for Δt from 1 to ntime
          if qtcounts[q_i][Δt] = 0 then continue
          for q_j from 1 to ncost
              probs[q_j][q_i][Δt] ← qqtcounts[q_j][q_i][Δt]/qtcounts[q_i][Δt]
  Smoothing: add half of smallest probability to all elements of probs with improving solution cost.
  // Normalize.
  for q_i from 1 to ncost
      for Δt from 1 to ntime
          sum ← 0
          for q_j from 1 to ncost
              sum ← sum + probs[q_j][q_i][Δt]
          for q_j from 1 to ncost
              probs[q_j][q_i][Δt] ← probs[q_j][q_i][Δt]/sum
  return probs
```

Figure 17: Pseudocode for profile estimation.

Figure 18 shows some of the plots created for the platform domain. Each plot has cost on the y axis and time on the x axis. Green circles represent inputs for which the policy says to keep searching, and red crosses represent inputs for which the policy says to stop

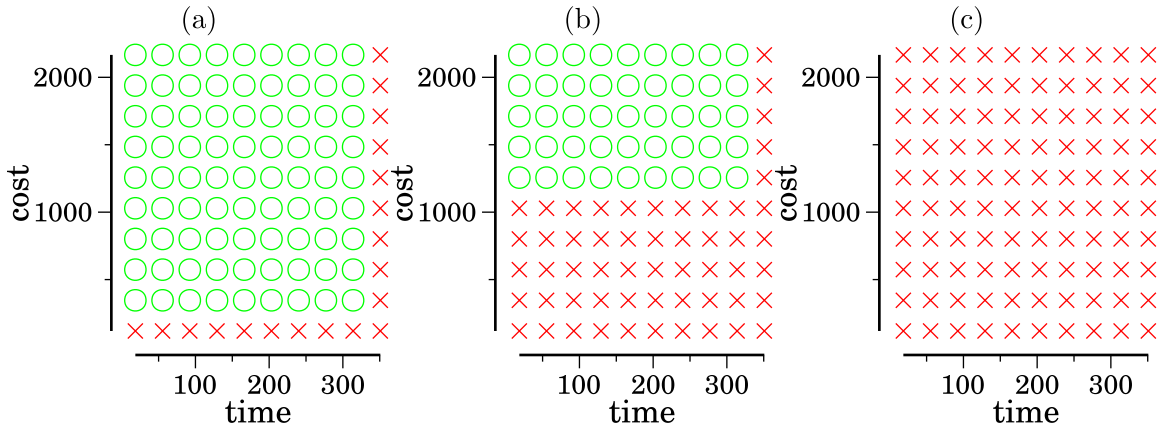

Figure 18: Three different policies: (a) prefers cheaper solutions at any expense ($w_f = 1, w_t = 0$), (b) attempts to trade some search time for some solution cost ($w_f = 0.6, w_t = 1$), and (c) prefers to have any solution as fast as possible ($w_f = 0, w_t = 1$).

searching and return the solution. As expected, the policy always continues when the goal is to minimize solution cost and always stops when the goal is to minimize search time (cf. the left-most and right-most plots in Figure 18 respectively). The center plot shows that we also successfully found policies that trade search time for solution cost by only stopping once the cost is sufficiently low. Finally, in the left-most plot, the bottom-most and right-most sides of the policy always stop as our implementation chose to stop when there was no training data available to estimate the profile for the given input values.

References

Björnsson, Y., Bulitko, V., & Sturtevant, N. (2009). TBA*: time-bounded A*. In *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI-09)*, pp. 431–436.

Boddy, M., & Dean, T. (1989). Solving time-dependent planning problems,. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, Vol. 2, pp. 979–984.

Burns, E., Hatem, M., Leighton, M. J., & Ruml, W. (2012). Implementing fast heuristic search code. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SoCS-12)*.

Burns, E., & Ruml, W. (2013). Iterative-deepening search with on-line tree size prediction. *Annals of Mathematics and Artificial Intelligence*, *S68*, 1–23.

Chen, P. C. (1992). Heuristic sampling: a method for predicting the performance of tree searching programs. *SIAM Journal on Computing*, *21*(2), 295–315.

Cox, M. T., & Raja, A. (2011). *Metareasoning: Thinking about thinking*. MIT Press.

Dean, T., & Boddy, M. (1988). An analysis of time-dependent planning. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, pp. 49–54.

Dean, T., Kaelbling, L. P., Kirman, J., & Nicholson, A. (1993). Planning with deadlines in stochastic domains. In *Proceedings of the eleventh national conference on Artificial intelligence*, Vol. 574, p. 579. Washington, DC.

Dechter, R., & Pearl, J. (1988). The optimality of A*. In Kanal, L., & Kumar, V. (Eds.), *Search in Artificial Intelligence*, pp. 166–199. Springer-Verlag.

Dionne, A. J., Thayer, J. T., & Ruml, W. (2011). Deadline-aware search using on-line measures of behavior. In *Proceedings of the Fourth Annual Symposium on Combinatorial Search (SoCS-11)*.

Domshlak, C., Karpas, E., & Markovitch, S. (2010). To max or not to max: Online learning for speeding up optimal planning. In *AAAI Conference on Artificial Intelligence (AAAI-10)*, pp. 1701–1706.

Dweighter, H. (1975). Elementary problem E2569. *American Mathematical Monthly*, *82*(10), 1010.

Finkelstein, L., & Markovitch, S. (2001). Optimal schedules for monitoring anytime algorithms. *Artificial Intelligence*, *126*(1), 63–108.

Garvey, A. J., & Lesser, V. R. (1993). Design-to-time real-time scheduling. *Systems, Man and Cybernetics, IEEE Transactions on*, *23*(6), 1491–1502.

Gates, W. H., & Papadimitriou, C. H. (1979). Bounds for sorting by prefix reversal. *Discrete Mathematics*, *27*(1), 47–57.

Hansen, E. A., & Zhou, R. (2007). Anytime heuristic search. *Journal of Artificial Intelligence Research*, *28*(1), 267–297.

Hansen, E. A., & Zilberstein, S. (2001). Monitoring and control of anytime algorithms: A dynamic programming approach. *Artificial Intelligence*, *126*, 139–157.

Hansen, E. A., Zilberstein, S., & Danilchenko, V. A. (1997). Anytime heuristic search: First results. Tech. rep., University Massachusetts, Amherst.

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

Helmert, M. (2010). Landmark heuristics for the pancake problem. In *Proceedings of the Third Symposium on Combinatorial Search (SoCS-10)*.

Helmert, M., & Röger, G. (2008). How good is almost perfect. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-08)*.

Hernández, C., Baier, J., Uras, T., & Koenig, S. (2012). Time-bounded adaptive A*. In *Proceedings of the Eleventh International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-12)*.

Horvitz, E., Ruan, Y., Gomes, C., Kautz, H., Selman, B., & Chickering, M. (2001). A Bayesian approach to tackling hard computational problems. In *Proceetings of the Seventeenth Conference on Uncertainty in Artificial Intelligence (UAI-01)*.

Kilby, P., Slaney, J., Thiébaux, S., & Walsh, T. (2006). Estimating search tree size. In *Proceedings of the twenty-first national conference on artificial intelligence (AAAI-06)*.

Knuth, D. E. (1975). Estimating the efficiency of backtrack programs. *Mathematics of computation*, *29*(129), 121–136.

Koenig, S., & Sun, X. (2009). Comparing real-time and incremental heuristic search for real-time situated agents. *Autonomous Agents and Multi-Agent Systems*, *18*(3), 313–341.

Korf, R. E. (1985). Iterative-deepening-A*: An optimal admissible tree search. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 1034–1036.

Korf, R. E. (1990). Real-time heuristic search. *Artificial intelligence*, *42*(2-3), 189–211.

Korf, R. E., Reid, M., & Edelkamp, S. (2001). Time complexity of iterative-deepening-A*. *Artificial Intelligence*, *129*(1), 199–218.

Likhachev, M., Gordon, G., & Thrun, S. (2003). ARA*: Anytime a* with provable bounds on sub-optimality. *Advances in Neural Information Processing Systems (NIPS-03)*, *16*.

Michie, D., & Ross, R. (1969). Experiments with the adaptive graph traverser. In *Machine Intelligence 5*, pp. 301–318.

Nilsson, N. J. (1969). A mobile automaton: an application of artificial intelligence techniques. In *Proceedings of the First International Joint Conference on Artificial intelligence (IJCAI-69)*, pp. 509–520.

Nilsson, N. J. (1980). *Principles of Artificial Intelligence*. Tioga Publishing Co.

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

Rice, J. R. (1976). The algorithm selection problem. *Advances in Computers*, *15*, 65–118.

Richter, S., Thayer, J. T., & Ruml, W. (2010). The joy of forgetting: Faster anytime search via restarting. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS-10)*, pp. 137–144.

Ruml, W., Do, M., Zhou, R., & Fromherz, M. P. (2011). On-line planning and scheduling: An application to controlling modular printers. *Journal of Artificial Intelligence Research*, *40*(1), 415–468.

Ruml, W., & Do, M. B. (2007). Best-first utility-guided search. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pp. 2378–2384.

Russell, S., & EricWefald (1991). *Do the right thing: studies in limited rationality*. The MIT Press.

Shekhar, S., & Dutta, S. (1989). Minimizing response times in real time planning and search. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pp. 238–242. Citeseer.

Sturtevant, N. (2012). Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, *4*(2), 144 – 148.

Sturtevant, N. R. (2011). Distance learning in agent-centered heuristic search. In *Proceedings of the Fourth Annual Symposium on Combinatorial Search (SoCS-11)*.

Thayer, J. (2012). *Faster Optimal and Suboptimal Heuristic Search*. Ph.D. thesis, University of New Hampshire.

Thayer, J. T., Benton, J., & Helmert, M. (2012). Better parameter-free anytime search by minimizing time between solutions. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SoCS-12)*.

Thayer, J. T., Dionne, A., & Ruml, W. (2011). Learning inadmissible heuristics during search. In *Proceedings of the Twenty-first International Conference on Automated Planning and Scheduling (ICAPS-11)*.

Thayer, J. T., & Ruml, W. (2008). Faster than weighted a*: An optimistic approach to bounded suboptimal search. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS-08)*.

Thayer, J. T., & Ruml, W. (2009). Using distance estimates in heuristic search. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS-09)*.

Thayer, J. T., & Ruml, W. (2010). Anytime heuristic search: Frameworks and algorithms. In *Proceedings of the Third Annual Symposium on Combinatorial Search (SoCS-10)*.

Tolpin, D., & Shimony, S. E. (2011). Rational deployment of CSP heuristics. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI-11)*.

Valenzano, R. A., Sturtevant, N., Schaeffer, J., Buro, K., & Kishimoto, A. (2010). Simultaneously searching with multiple settings: An alternative to parameter tuning for suboptimal single-agent search algorithms. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS-10)*.

van den Berg, J., Shah, R., Huang, A., & Goldberg, K. (2011). ANA*: Anytime nonparametric A*. In *Proceedings of Twenty-fifth AAAI Conference on Artificial Intelligence (AAAI-11)*.

Zohavi, U., Felner, A., Burch, N., & Holte, R. (2010). Predicting the performance of ida* using conditional distributions. *Journal of Artificial Intelligence Research*, *37*(1), 41–84.