Best-first Utility-guided Search

Wheeler Ruml
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304 USA
ruml at parc dot com

Elisabeth H. Crawford
Computer Science Department
Carnegie Mellon University
Pittsburgh PA 15213 USA
ehc at cs.cmu dot edu

Abstract

In many shortest-path problems of practical interest, insufficient time is available to find a provably optimal solution. In dynamic environments, for example, the expected value of a plan may decrease with the time required to find it. One can only hope to achieve an appropriate balance between search time and the resulting plan cost. Several algorithms have been proposed for this setting, including weighted A*, Anytime A*, and ARA*. These algorithms multiply the heuristic evaluation of a node, exaggerating the effect of the cost-to-go. We propose a more direct approach, called BUGSY, in which one explicitly estimates search-nodes-to-go. One can then attempt to optimize the overall utility of the solution, expressed by the user as a function of search time and solution cost. Experiments in several problem domains, including motion planning and sequence alignment, demonstrate that this direct approach can surpass anytime algorithms without requiring performance profiling.

1 Introduction

Many important tasks, such as planning, parsing, and sequence alignment, can be represented as shortest-path problems. If sufficient computation is available, optimal solutions to such problems can be found using A* search with an admissible heuristic [Hart *et al.*, 1968]. However, in many practical scenarios, time is limited or costly and it is not desirable, or even feasible, to look for the least-cost path. Furthermore, in dynamic environments, a plan's chance of becoming invalid increases with time, making any plan based on current knowledge less valuable as time passes. Instead of ensuring an optimal solution, search effort should be carefully allocated in a way that balances the cost of the paths found with the required computation time. This trade-off is expressed by the user's utility function, which specifies the subjective value of every combination of solution quality and search time. In this paper, we introduce a new shortest-path algorithm called BUGSY that explicitly acknowledges the user's utility function and uses it to guide its search.

A* is a best-first search in which the 'open list' of unexplored nodes is sorted by $f(n) = g(n) + h(n)$, where $g(n)$ denotes the known cost of reaching a node n from the initial state and $h(n)$ is typically a lower bound on the cost of reaching a solution from n. A* is optimal in the sense that no algorithm that returns an optimal solution using the same lower bound function $h(n)$ visits fewer nodes [Dechter and Pearl, 1988]. However, in many applications solutions are needed faster than A* can provide them. To find a solution faster, it is common practice to increase the weight of $h(n)$ via $f(n) = g(n) + w \cdot h(n)$, with $w \geq 1$ [Pohl, 1970]. There are many variants of weighted A* search, including A^*_ϵ [Pearl and Kim, 1982], Anytime A* [Hansen *et al.*, 1997; Zhou and Hansen, 2002], and ARA* [Likhachev *et al.*, 2004]. In ARA*, for example, a series of solutions of decreasing cost is returned over time. The weight w is initially set to a high value and then decremented by δ after each solution. If allowed to continue, w eventually reaches 1 and the cheapest path is discovered. Of course, finding the optimal solution this way takes longer than simply running A* directly.

These algorithms suffer from two inherent difficulties. First, it is not well understood how to set w or δ to best satisfy the user's needs. Setting w too high or δ too low can result in many poor-quality solutions being returned, wasting time. But if w is set too low or δ too high, the algorithm may take a very long time to find a solution. Therefore, to use a weighted A* technique like ARA* the user must perform many pilot experiments in each new problem domain to find good parameter settings.

Second, for anytime algorithms such as ARA*, the user must estimate the right time to stop the algorithm. The search process appears as a black box that could emit a significantly better solution at any moment, so one must repeatedly estimate the probability that continuing the computation will be worthwhile according to the user's utility function. This requires substantial prior statistical knowledge of the run-time performance profile of the algorithm and rests on the assumption that such learned knowledge applies to the current instance.

These difficulties point to a more general problem: anytime algorithms must inherently provide suboptimal performance due to their ignorance of the user's utility function. It is simply not possible in general for an algorithm to quickly transform the best solution achievable from scratch in time t into the best solution achievable in time $t + 1$. In the worst case, visiting the next-most-promising solution might require

starting back at a child of the root node. Without the ability to decide during the search whether a distant solution is worth the expected effort of reaching it, anytime algorithms must be manually engineered according to a policy fixed in advance. Such hardcoded policies mean that there will inevitably be situations in which anytime algorithms will either waste time finding nearby poor-quality solutions or overexert themselves finding a very high quality solution when any would have sufficed.

In this paper we address the fundamental issue: knowledge of the user's utility function. We propose a simple variant of best-first search that represents the user's desires and uses an estimate of this utility as guidance. We call the approach BUGSY (Best-first Utility-Guided Search—Yes!) and show empirically across several domains that it can successfully adapt its behavior to suit the user, sometimes significantly outperforming anytime algorithms. Furthermore, this utility-based methodology is easy to apply, requiring no performance profiling.

2 The BUGSY Approach

Ideally, a rational search agent would evaluate the utility to be gained by each possible node expansion. The utility of an expansion is equal to the utility of the eventual outcomes enabled by that expansion, namely the solutions lying below that node. For instance, if there is only one solution in a tree-structured space, expanding any node other than the one it lies beneath has no utility (or negative utility if time is costly). We will approximate these true utilities by assuming that the utility of an expansion is merely the utility of the highest-utility solution lying below that node.

We will further assume that the user's utility function can be captured in a simple linear form. If $f(s)$ represents the cost of solution s, and $t(s)$ represents the time at which it is returned to the user, then we expect the user to supply three constants: $U_{default}$, representing the utility of returning an empty solution; w_f, representing the importance of solution quality; and w_t, representing the importance of computation time. The utility of expanding node n is then computed as

$$U(n) = U_{default} - \min_{s \text{ under } n} (w_f \cdot f(s) + w_t \cdot t(s))$$

where s ranges over the possible solutions available under n. (Note that we follow the decision-theoretic tradition of better utilities being more positive, requiring us to subtract the estimated solution cost $f(s)$ and search time $t(s)$.) This formulation allows us to express exclusive attention to either cost or time, or any linear trade-off between them. 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 utility function would also be necessary when constructing the termination policy for an anytime algorithm.) Although superficially similar to weighted A*, BUGSY's node evaluation function differs because w_f is applied to both $g(n)$ and $h(n)$.

Of course, the solutions s available under a node are unknown, but we can estimate some of their utilities by using functions analogous to the traditional heuristic function $h(n)$. Instead of merely computing a lower bound on the cost of the cheapest solution under a node, we also compute the lower bound on distance in search nodes to that hypothetical cheapest solution. In many domains, this additional estimate entails only trivial modifications to the usual h function. Search distance can then be multiplied by an estimate of time per expansion to arrive at $t(s)$. (Note that this simple estimation method makes the standard assumption of constant time per node expansion.) To provide a more informed estimate, we can also compute bounds on the cost and time to the nearest solution in addition to the cheapest. $U(n)$ can then be estimated as the maximum of the two utilities. For convenience, we will also notate by $f(n)$ and $t(n)$ the values inherited from whichever hypothesized solution had the higher utility.

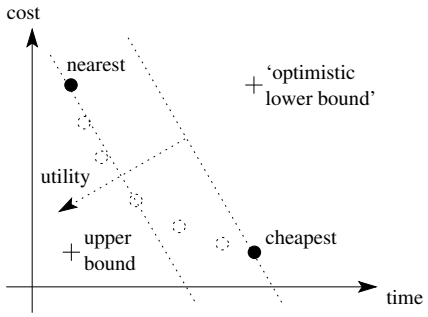


Figure 1: Estimating utility using the maximum of bounds on the nearest and cheapest solutions.

Figure 1 illustrates this process. The two solid dots represent the solutions hypothesized by the cheapest and nearest heuristic functions. The dashed circles represent hypothetical solutions representing a trade-off between those two extremes. The dotted lines represent contours of constant utility and the dotted arrow shows the direction of the utility gradient. Assuming that the two solid dots represent lower bounds, then an upper bound on utility would combine the cost of the cheapest solution with the time to the nearest solution. However, this is probably a significant overestimate. Taking the time of the cheapest and the cost of the nearest is not a true lower bound on utility because the two hypothesized solutions are themselves lower bounds and might in reality lie further toward the top and right of the figure. Note that under different utility functions (different slopes for the dotted lines) the relative superiority of the nearest and cheapest solutions can change.

2.1 Implementation

Figure 2 gives a pseudo-code sketch of a BUGSY implementation. The algorithm closely follows a standard best-first search. $U(n)$ is an estimate, not a lower bound, so it can overestimate or change arbitrarily along a path. This implies that we might discover a better route to a previously expanded state. Duplicate paths to the same search state are detected in steps 7 and 10; only the cheaper path is retained. We record links to a node's children as well as the preferred parent so that the utility of descendants can be recomputed (step 9) if

BUGSY(*initial,* $U()$)
1. *open* $\leftarrow$ {*initial*}, *closed* $\leftarrow$ {}
2. $n \leftarrow$ remove node from *open* with highest $U(n)$ value
3. if n is a goal, return it
4. add n to *closed*
5. for each of n's children c,
6. if c is not a goal and $U(c) < 0$, skip c
7. if an old version of c is in *closed*,
8. if c is better than c_{old},
9. update c_{old} and its children
10. else, if an old version of c is in *open*,
11. if c is better than c_{old},
12. update c_{old}
13. else, add c to *open*
14. go to step 2

Figure 2: BUGSY follows the outline of best-first search.

$g(n)$ changes [Nilsson, 1980, p. 66]. The on-line estimation of time per expansion has been omitted for clarity. The exact ordering function used for *open* (and to determine 'better' in steps 8 and 11) prefers high $U(n)$ values, breaking ties for low $t(n)$, breaking ties for low $f(n)$, breaking ties for high $g(n)$. Note that the linear formulation of utility means that *open* need not be resorted as time passes because all nodes lose utility at the same constant rate independent of their estimated solution cost. In effect, utilities are stored independent of the search time so far.

The $h(n)$ and $t(n)$ functions used by BUGSY do not have to be lower bounds. BUGSY requires estimates—there is no admissibility requirement. If one has data from previous runs on similar problems, this information can be used to convert standard lower bounds into estimates [Russell and Wefald, 1991]. In the experiments reported below, we eschew the assumption that training data is available and compute corrections on-line. We keep a running average of the one-step error in the cost-to-go and distance-to-go, measured at each node generation. These errors are computed by comparing the cost-to-go and distance-to-go of a node with those of its children. If the cost-to-go has not decreased by the cost of the operator used to generate the child, we can conclude that the parent's value was too low and record the discrepancy as an error. Similarly, the distance-to-go should have decreased by one. These correction factors are then used when computing a node's utility to give a more accurate estimate based on the experience during the search so far. Given the raw cost-to-go value h and distance-to-go value d and average errors e_h and e_d, $d' = d(1 + e_d)$ and $h' = h + d'e_h$. Because on-line estimation of the time per expansion and the cost and distance corrections create additional overhead for BUGSY relative to other search algorithms, we will take care to measure CPU time in our experimental evaluation, not just node generations.

2.2 Properties of the Algorithm

BUGSY is trivially sound—it only returns nodes that are goals. If the heuristic and distance functions are used without inadmissible corrections, then the algorithm is also complete if the search space is finite. If $w_t = 0$ and $w_f > 0$, BUGSY reduces to A*, returning the cheapest solution. If $w_f = 0$ and $w_t > 0$, then BUGSY is greedy on $t(n)$. Ties will be broken on low $f(n)$, so a longer route to a previously visited state will be discarded. This limits the size of *open* to the size of the search space, implying that a solution will eventually be discovered. Similarly, if both w_f and $w_t > 0$, BUGSY is complete because $t(n)$ is static at every state. The $f(n)$ term in $U(n)$ will then cause a longer path to any previously visited state to be discarded, bounding the search space and ensuring completeness. Unfortunately, if the search space is infinite and $w_t > 0$, BUGSY is not complete because a pathological $t(n)$ can potentially mislead the search forever.

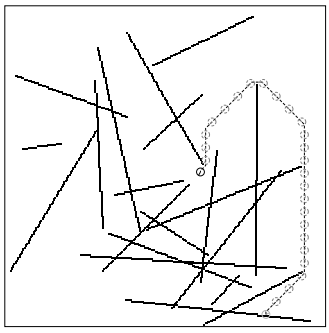
If the utility estimates $U(n)$ are perfect, BUGSY is optimal. This follows because it will proceed directly to the highest-utility solution. Assuming $U(n)$ is perfect, when BUGSY expands the start node the child node on the path to the highest utility solution will be put at the front of the open list. BUGSY will expand this node next. One of the children of this node must have the highest utility on the open list since it is one step closer to the goal than its parent, which previously had the highest utility, and it leads to a solution of the same quality. In this way, BUGSY proceeds directly to the highest utility solution achievable from the start state. It incurs no loss in utility due to wasted time since it only expands nodes on the path to the optimal solution.

It seems intuitive that BUGSY might have application in problems where operators have different costs and hence the distance to a goal in the search space might not correspond directly to its cost. But even in a search space in which all operators have unit cost (and hence the nearest and cheapest heuristics are the same), BUGSY can make different choices than A*. Consider a situation in which, after several expansions, it appears that node A, although closer to a goal than node B, might result in a worse overall solution. (Such a situation can easily come about even with an admissible and consistent heuristic function.) If time is weighted more heavily than solution cost, BUGSY will expand node A in an attempt to capitalize on previous search effort and reach a goal quickly. A*, on the other hand, will always abandon that search path and expand node B in a dogged attempt to optimize solution cost regardless of time.

In domains in which the cost-to-goal and distance-to-goal functions are different, BUGSY can have a significant advantage over weighted A*. With a very high weight, weighted A* will find a solution only as quickly as the greedy algorithm. BUGSY however, because its search is guided by an estimate of the distance to solutions as well as their cost, can actually find a solution in less time than the greedy algorithm.

3 Empirical Evaluation

To determine whether such a simple mechanism for time-aware search can be effective in practice with imperfect estimates of utility, we compared BUGSY against seven other algorithms on three different domains: gridworld path planning (12 different varieties), dynamic robot motion planning (used by Likhachev *et al.* [2004] to evaluate ARA*), and multiple sequence alignment (used by Zhou and Hansen [2002] to



```
# # ##   #
  oooooooo
  o #     o
  o     # o
  o   #   o
##o#     #o
Soo#      G


jhgiggh
j-grmgg
o-grm-o
```

Figure 3: Examples of the test domains: dynamic motion planning (left), gridworld planning (top right), and multiple sequence alignment (bottom right).

evaluate Anytime A*). All algorithms were coded in Objective Caml, compiled to native code, and run on one processor of a dual 2.6GHz Xeon machine with 2Gb RAM, measuring CPU time used. The algorithms were:

A* detecting duplicates using a closed list, breaking ties on f in favor of high g,

weighted A* with $w = 3$,

greedy A* but preferring low h, breaking ties on low g,

speedy greedy but preferring low time to goal ($t(n)$), breaking ties on low h, then low g,

Anytime A* weighted A* ($w = 3$) that continues, pruning the open list, until an optimal goal has been found,

ARA* performs a series of weighted A* searches (starting with $w = 3$), decrementing the weight ($\delta = 0.2$, following Likhachev et al.) and reusing search effort,

A^*_ϵ from among those nodes within a factor of ϵ (3) of the lowest f value in the open list, expands the one estimated to be closest to the goal.

Note that greedy, speedy, and A* do not provide any inherent mechanism for adjusting their built-in trade-off of solution cost against search time; they are included only to provide a frame of reference for the other algorithms. The first solution found by Anytime A* and ARA* is the same one found by weighted A*, so those algorithms should do at least as well. We confirmed this experimentally, and omit weighted A* from our presentation below. On domains with many solutions, Anytime A* often reported thousands of solutions; we therefore limited both anytime algorithms to only reporting solutions that improve solution quality by at least 0.1%. A^*_ϵ performed very poorly in our preliminary tests, taking a very long time, so we omit its results as well. [1]

3.1 Dynamic Robot Motion Planning

Following Likhachev *et al.* [2004], this domain involves motion planning for a mobile robot (see Figure 3 for an example). Rather than finding the shortest path, the objective is to find the fastest path, taking into account the maximum acceleration of the robot and its inability to turn quickly at high speed. Solution cost corresponds to the duration of the planned robot trajectory. In effect, each utility function specifies a different trade-off between planning time and plan execution time. The state representation records position, heading, and speed. The path cost heuristic ($h(n)$) is simply the shortest path distance to the goal, divided by the maximum speed. This is precomputed to all cells at the start of the search. The plan cost lower bound $f(n)$ is the usual cost-so-far ($g(n)$) plus this cost-to-go ($h(n)$). For speedy and BUGSY, the distance in moves to the goal is also precomputed. The search cost estimate $t(n)$ is this distance divided by the number of search nodes expanded per second, which was estimated on-line as discussed above. No separate estimates were made for BUGSY of the distance to the cheapest goal or cost of the nearest goal, so U was estimated only on this single f and t values. Legal state transitions (ignoring position) were precomputed. Unlike the heuristics, this was the same for all algorithms and was not included in the search time. We used 20 worlds 100 by 100 meters (discretized as in Likhachev et al. every 0.4 meters), each with 20 linear obstacles placed at random. Starting and goal positions and headings were selected uniformly at random. Instances that were solved by A* in less than 10 seconds or more than 1000 seconds were replaced.

$U()$	BUGSY	ARA*	Sp	Gr
time only	72	66	75	88
10 microsec	72	66	75	88
100 microsec	69	66	74	88
1 msec	58	63	70	83
10 msec	51	47	47	56
0.1 sec	66	59	53	55
1 sec	69	65	56	56
10 secs	67	69	53	54
100 secs	67	69	53	53

Table 1: Results on dynamic robot motion planning.

Table 1 compares the solutions obtained by each algorithm under a range of different possible utility functions. Each row of the table corresponds to a different utility function. Recall that each utility function is a weighted combination of path cost and CPU time taken to find it. The relative size of the weights determines how important time is relative to cost. In other words, the utility function specifies the maximum amount of time that should be spent to gain an improvement of 1 cost unit. This is the time that is listed under U() for each row in the table. For example, "1 msec" means that a solution that takes 0.001 seconds longer to find than another must be at least 1 unit cheaper to be judged superior. The utility functions tested range over several orders of magnitude from one in which only search time matters to one in which 100 seconds can be spent to obtain a one unit improvement in the solution cost.

Recall that, given a utility function at the start of its search, BUGSY returns a single solution representing the best trade-

[1] Although Pearl and Kim do not discuss implementation techniques (their results are presented solely in terms of node expansions), it seems that their algorithm could be made to operate more efficiently by designing a special coordinated heap and balanced binary tree data structure. We have not pursued this yet.

off of path cost and search time that it could find based on the information available to it. Of course, the CPU time taken is recorded along with the solution cost. Greedy (notated Gr in the table) and speedy (notated Sp) also each return one solution. These solutions may score well according to utility functions with extreme emphasis on time but may well score poorly in general. The two anytime algorithms, Anytime A* and ARA*, return a stream of solutions over time. For these experiments, we allowed them to run to optimality and then, for each utility function, post-processed the results to find the optimal cut-off time to optimize each algorithm's performance for that utility function. Note that this 'clairvoyant termination policy' gives Anytime A* and ARA* an unrealistic advantage in our tests. However, both A* and Anytime A* performed extremely poorly in this domain and are omitted from Table 1. To compare more easily across different utility functions, all of the resulting solution utilities were linearly scaled to fall between 0 and 100. Each cell in the table is the mean across 20 instances.

The results suggest that BUGSY is competitive with or better than ARA* on all but perhaps one of the utility functions. In general, BUGSY seems to offer a slight advantage when time is important. Given that BUGSY does not require performance profiling to construct a termination policy, this is encouraging performance. As one might expect, Greedy performs well when time is very important, however as cost becomes important the greedy solution is less useful. Compared to greedy, speedy is not able to overcome the overhead of computing two node evaluation functions.

3.2 Gridworld Planning

We considered several classes of path planning problems on a 500 by 300 grid, using either 4-way or 8-way movement, three different probabilities of blocked cells, and two different cost functions. In addition to unit costs, under which every move is equally expensive, we used a graduated cost function in which moves along the upper row are free and the cost goes up by one for each lower row. Figure 3 shows a small example solution under these costs (the start and goal positions are always in these corners). We call this cost function 'life' because it shares with everyday living the property that a short direct solution that can be found quickly (shallow in the search tree) is relatively expensive while a least-cost solution plan involves many annoying economizing steps. Under both cost functions, simple analytical lower bounds (ignoring obstacles) are available for the cost ($g(n)$) and distance (in search steps) to the cheapest goal and to the nearest goal. These quantities are then used to compute the $f(n)$ and $t(n)$ estimates. Because A* can perform well in this domain and our experiments include utility functions that make it worth finding the optimal solution, we diluted BUGSY's estimated lower-bound correction factors by dividing them by 5, decreasing the severity of any overestimation.

Table 2 shows typical results from three representative classes of gridworld problems. As before, the rows represent a broad spectrum of utility functions, including those in which speedy and A* are each designed to be optimal. Each value represents the mean over 20 instances. Anytime A* is notated AA*. In the top group (unit costs, 8-way movement, 40% blocked), we see BUGSY performing very well, behaving like speedy and greedy when time is important, like A* when cost is important, and significantly surpassing all the algorithms for the middle range of utility functions. In the next group (4-way movement, 20% blocked), BUGSY performs very well as long as time has some importance, again dominating in the middle range of utility functions where balancing time and cost is crucial. However, its inadmissible heuristic means that it cannot perform quite as well as A* or ARA* at the edge of the spectrum when cost becomes critical. (Of course, one can always disable BUGSY's correction factors when running under such circumstances, but presumably in practice one would be using A* search anyway if search time weren't an important consideration.) In the bottom group in the table ('life' costs, 4-way movement, 20% blocked), we see a similar general pattern: BUGSY performs very well across a wide range of utility functions, dominating other algorithms for the middle range of utility functions. However, it does fall slightly short of A* when solution cost is the only criterion.

$U()$	BUGSY	ARA*	AA*	Sp	Gr	A*
	unit costs, 8-way movement, 40% blocked					
time only	99	100	99	99	100	69
500 microsec	98	96	96	95	95	69
1 msec	98	91	93	90	91	69
5 msec	95	60	68	56	56	68
10 msec	94	44	57	34	34	74
50 msec	95	85	77	33	33	91
cost only	95	96	96	33	33	96
	unit costs, 4-way movement, 20% blocked					
time only	97	98	98	98	99	19
100 microsec	95	94	95	94	95	21
500 microsec	91	67	70	61	62	28
1 msec	86	62	43	28	29	50
5 msec	82	81	42	22	22	91
10 msec	79	87	46	20	20	92
cost only	76	93	93	19	19	93
	'life' costs, 4-way movement, 20% blocked					
time only	99	92	88	100	96	16
1 microsec	97	94	90	93	98	17
5 microsec	92	89	85	52	92	18
10 microsec	93	86	83	12	88	30
50 microsec	97	86	87	11	85	87
100 microsec	97	91	89	11	85	94
cost only	94	97	97	11	82	97

Table 2: Results on three varieties of gridworld planning.

3.3 Multiple Sequence Alignment

Alignment of multiple strings has recently been a popular domain for heuristic search algorithms [Hohwald *et al.*, 2003]. An example alignment is shown in Figure 3. The state representation is the number of characters consumed so far from each string; a goal is reached when all characters are consumed. Moves that consume from only some of the strings represent the insertion of a 'gap' character into the others. We computed alignments of three sequences at a time, using the

$U()$	BUGSY	ARA*	AA*	Sp	Gr	A*
time only	99	100	100	100	100	22
1 msec	100	99	99	97	98	22
5 msec	99	97	97	88	92	24
10 msec	98	92	94	74	83	26
50 msec	87	80	81	14	42	90
0.1 sec	69	89	68	11	33	93
cost only	57	95	95	9	27	95

Table 3: Results on multiple sequence alignment.

standard 'sum-of-pairs' cost function in which a gap costs 2, a substitution (mismatched non-gap characters) costs 1, and costs are computed by summing all the pairwise alignments. Sequences were over 20 characters, representing amino acid triplets. The uniform random sequences that are popular benchmarks for optimal alignment algorithms are not suitable in our setting because the solution found by the speedy algorithm (merely traversing the diagonal, resulting in many substitutions) is very often the optimal alignment. Instead, we use biologically-inspired benchmarks which encourage optimal solutions that contain significant numbers of gaps and matches. Starting from a 'common ancestor' string which does not become part of the instance, we create sequences by deleting and substituting characters uniformly at random. In the instances used below, the ancestors were 1000 characters long and the probabilities of deletion and substitution were both 0.25 at each position. The heuristic function $h(n)$ was based on optimal pairwise alignments that were precomputed by dynamic programming. The lower bound on search nodes to go was simply the maximum number of characters remaining in any sequence. As in gridworld, A* is a feasible algorithm and thus we dilute BUGSY's correction factors by 5.

Table 3 shows the results, with each row representing a different utility function and all raw scores again normalized between 0 and 100. Each cells represents the mean over 5 instances (there was little variance in the scores in this domain). Again we see the same pattern of performance. BUGSY performs very well when time is important and surpasses the other algorithms when balancing between cost and time. It does fall short of A* when cost is paramount, due to its inadmissible heuristic.

4 Discussion

We have presented empirical results, using actual CPU time measurements and a variety of search problems, demonstrating that BUGSY is at least competitive with state-of-the-art anytime algorithms. For utility functions with an emphasis on solution time or on balancing time and cost, it often performs significantly better than any previous method. However, for utility functions based heavily on solution cost it can sometimes perform worse than A*. BUGSY appears quite robust across different domains and utility functions.

When its utility estimates are perfect, BUGSY is optimal. However, more work remains to understand the exact tradeoff between accuracy and admissibility. Our empirical experience demonstrates that attempting to correct lower bounds into more accurate estimators can impair BUGSY's performance when solution quality is very important. However, it seems foolish not to take advantage of on-line error estimation to bring these bounds closer to the accurate estimates that would allow BUGSY to be optimal. In this paper, we have chosen to merely dilute the correction factors. In the future, we hope to be able to analyze the given utility function in the context of the domain and determine whether admissibility is worth preserving.

We have done preliminary experiments incorporating simple deadlines into BUGSY, with encouraging results. Because it estimates the search time-to-go, it can effectively prune solutions that lie beyond a search time deadline. Another similar extension applies to temporal planning: one can specify a bound on the sum of the search time and the resulting plan's execution time and let BUGSY determine how to allocate the time.

Note that BUGSY solves a different problem than Real-Time A* [Korf, 1990] and its variants. Rather than performing a time-limited search for the first step in a plan, BUGSY tries to find a complete plan to a goal in limited time. This is particularly useful in domains in which operators are not invertible or are otherwise costly to undo. Having a complete path to a goal ensures that execution does not become ensnared in a deadend. It is also a common requirement when planning is but the first step in a series of computations that might further refine the action sequence.

In some applications of best-first search, memory use is a prominent concern. In a time-bounded setting this is less frequently a problem because the search doesn't have time to exhaust available memory. However, the simplicity of BUGSY means that it may well be possible to integrate some of the techniques that have been developed to reduce the memory consumption of best-first search if necessary.

When planning in a dynamic environment, we assume not only that BUGSY is provided with a utility function that captures the decrease in expected plan value as a linear function of time, but also that the algorithm has full access to knowledge of how the domain changes. It would be very interesting to combine the utility-based search of BUGSY with techniques to exploit localized changes in the search space, such as used in ARA*.

5 Conclusions

As Nilsson notes, "in most practical problems we are interested in minimizing some *combination* of the cost of the path and the cost of the search required to obtain the path" yet "combination costs are never actually computed ... because it is difficult to decide on the way to combine path cost and search-effort cost" [1971, p. 54, emphasis his]. BUGSY addresses this problem by letting the user specify how path cost and search cost should be combined.

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 on the information they collect so as to directly maximize the user's utility.

References

[Dechter and Pearl, 1988] Rina Dechter and Judea Pearl. The optimality of A*. In Laveen Kanal and Vipin Kumar, editors, *Search in Artificial Intelligence*, pages 166–199. Springer-Verlag, 1988.

[Hansen *et al.*, 1997] Eric A. Hansen, Shlomo Zilberstein, and Victor A. Danilchenko. Anytime heuristic search: First results. CMPSCI 97-50, University of Massachusetts, Amherst, September 1997.

[Hart *et al.*, 1968] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics*, SSC-4(2):100–107, July 1968.

[Hohwald *et al.*, 2003] Heath Hohwald, Ignacio Thayer, and Richard E. Korf. Comparing best-first search and dynamic programming for optimal multiple sequence alignment. In *Proceedings of IJCAI-03*, pages 1239–1245, 2003.

[Korf, 1990] Richard E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42:189–211, 1990.

[Likhachev *et al.*, 2004] Maxim Likhachev, Geoff Gordon, and Sebastian Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Proceedings of NIPS 16*, 2004.

[Nilsson, 1971] Nils J. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, 1971.

[Nilsson, 1980] Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Co, 1980.

[Pearl and Kim, 1982] Judea Pearl and Jin H. Kim. Studies in semi-admissible heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4(4):391–399, July 1982.

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

[Russell and Wefald, 1991] Stuart Russell and Eric Wefald. *Do the Right Thing: Studies in Limited Rationality*. MIT Press, 1991.

[Zhou and Hansen, 2002] Rong Zhou and Eric A. Hansen. Multiple sequence alignment using anytime A*. In *Proceedings of AAAI-02*, pages 975–976, 2002.