On the Benefits of Randomly Adjusting Anytime Weighted A*

Abhinav Bhatia, Justin Svegliato, Shlomo Zilberstein
College of Information and Computer Sciences
University of Massachusetts Amherst
{abhinavbhati, jsvegliato, shlomo}@cs.umass.edu

Abstract

Anytime Weighted A*—an anytime heuristic search algorithm that uses a weight to scale the heuristic value of each node in the open list—has proven to be an effective way to manage the trade-off between solution quality and computation time in heuristic search. Finding the best weight, however, is challenging because it depends on not only the characteristics of the domain and the details of the instance at hand, but also the available computation time. We propose a randomized version of this algorithm, called *Randomized Weighted A**, that randomly adjusts its weight at runtime and show a counterintuitive phenomenon: RWA* generally performs as well or better than AWA* with the best static weight on a range of benchmark problems. The result is a simple algorithm that is easy to implement and performs consistently well without any offline experimentation or parameter tuning.

Introduction

Anytime Weighted A* (AWA*), which is an anytime variant of the heuristic search algorithm A*, uses a numeric weight to scale the heuristic value of each node in the open list to alter the order of node expansion. Intuitively, higher weights make the algorithm more "greedy" with respect to low-hanging solutions while potentially compromising total solution quality (Pohl 1970). The ability to control the trade-off between runtime and solution quality has made AWA* useful for a wide range of complex problems, such as multiple sequence alignment (Kobayashi and Imai 1998; Zhou and Hansen 2002), path planning (Likhachev, Gordon, and Thrun 2003), and robotic arm motion planning (Cohen, Chitta, and Likhachev 2014). Finding the best weight, however, is challenging since it depends not only on the characteristics of the domain and the details of the instance at hand but also on the available computation time.

Early work on AWA* has focused on selecting the best static weight for a problem (Hansen and Zhou 2007), changing the weight for an instance of a problem (Sun, Druzdzel, and Yuan 2007), or adjusting the weight at runtime heuristically (Thayer and Ruml 2009). It has also been observed that AWA* can be improved via restarting when a solution is found (Richter, Thayer, and Ruml 2010). Wilt and Ruml (2012) analyzed the failure conditions of AWA* with respect to its weight. These methods have shown that finding the best static weight for a problem is laborious as it involves exhaustive evaluation of a range of weights on sample problem instances. Efforts to dynamically modify the weight based on the progress made by the algorithm and the available computation time have shown promising but limited success (Thayer and Ruml 2008), leaving the question of how to optimize the weight in a principled way an open problem.

We propose a randomized variant of AWA*, *Randomized Weighted A** (RWA*), that adjusts its weight randomly at runtime. We show a counterintuitive phenomenon: RWA* performs as well or better than AWA* on a range of benchmark problems given a deadline. First, RWA* computes solutions with a higher quality on average compared to AWA* with any static weight. Second, RWA* computes solutions at least as good as AWA* with any static weight on more than 60% of the instances of a problem. This shows the benefits of using different weights at runtime instead of a single static weight for an instance of a problem. Third, RWA* has a higher probability of computing at least one solution compared to AWA* with any static weight. Fourth, RWA* is easy to implement as it does not require any extensive offline experimentation or parameter tuning to find the best weight.

The paper proceeds as follows. Section 2 reviews AWA* and proposes RWA* that adjusts its weight randomly at runtime. Section 3 evaluates RWA* by showing that it is more robust than AWA* with any static weight on a range of benchmark problems. Section 4 concludes the paper.

Randomly Adjusting Anytime Weighted A*

AWA* is an anytime heuristic search algorithm based on four design principles: it (1) uses an inadmissible heuristic function to find suboptimal solutions, (2) continues the heuristic search after each suboptimal solution is found, (3) provides an error bound on a suboptimal solution when it is interrupted, and (4) guarantees an optimal solution once the open list is empty (Hansen, Zilberstein, and Danilchenko 1997; Hansen and Zhou 2007). This algorithm has led to many related heuristic search algorithms (Thayer and Ruml 2010): *anytime repairing A** that tunes its suboptimal bound based on the available computation time (Likhachev, Gordon, and Thrun 2003; Likhachev et al. 2008), *anytime window A** that expands nodes within a sliding window of the search tree (Aine, Chakrabarti, and Kumar 2007), and *any-*

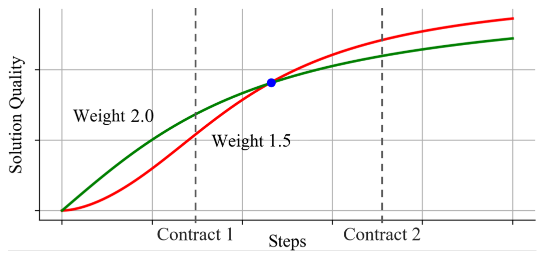

Figure 1: An example of two executions of anytime A*.

*time nonparametric A** that avoids adjustable parameters altogether (Van Den Berg et al. 2011).

Most importantly, AWA* replaces the standard evaluation function $f(n) = g(n) + h(n)$ used to select the next node for expansion from the open list with a weighted evaluation function $f_w(n) = g(n) + w \cdot h(n)$, where the path cost function $g(n)$ represents the cost of the path from the start node to a given node n and $h(n)$ represents an admissible heuristic function that estimates the cost from a given node n to the goal node. Given a weight $w > 1$, the weighted heuristic becomes potentially inadmissible and AWA* prioritizes expanding a node that appears closer to a goal node by weighting the heuristic component $h(n)$ more heavily than the path cost component $g(n)$. This causes AWA* to decrease the computation time of solutions potentially at the expense of quality, which is inversely proportional to cost.

Fig. 1 shows typical performance curves for two executions of AWA* that solve an instance of a problem with different weights: a higher weight of 2.0 leads to better expected quality in the short term as it finds solutions more quickly at the expense of long term quality while a lower weight of 1.5 leads to better expected quality in the long term as it finds solutions more slowly but emphasizes quality. This raises the question of how to select the weight of AWA* to control the trade-off between solution quality and computation time during execution in a contract setting where an anytime algorithm is interrupted at a known deadline and the current solution is returned (Zilberstein 1996).

We propose a randomized version of AWA*, called *Randomized Weighted A** (RWA*), that is based on AWA* (Zhou and Hansen 2002) with several simple modifications. Before each node expansion, RWA* adjusts the weight w by sampling it uniformly from a set of weights $\mathcal{W} = \{w_1, w_2, \dots, w_\ell\}$ to modify expansion priorities of the open nodes that are on the frontier. The algorithm maintains ℓ open lists, one for each weight, in order to avoid having to reorder the nodes if it were to maintain just one open list. The ℓ open lists offer different orderings of the same set of nodes. RWA* inserts every new node into or deletes a given node from the ℓ open lists, each represented as a min heap ordered according to the f_w-value of a specific weight $w \in \mathcal{W}$. While the worst-case time complexity for inserting or deleting a node across all ℓ open lists of size n sequentially is $\mathcal{O}(\ell \log n)$, these operations can be performed in parallel to retain a worst-case time complexity of $\mathcal{O}(\log n)$. Similarly, for the worst-case space complexity, all heaps only store references to the same set of nodes to retain a worst-case space complexity of $\mathcal{O}(n)$.

Algorithm 1 RWA* — Randomized Weighted A*

```
Require: A start node $n_0$, a set of weights $\mathcal{W}$
1: procedure RWA*($n_0, \mathcal{W}$)
2:     for all $w \in \{1\} \cup \mathcal{W}$ do
3:         $Q_w \leftarrow$ MINHEAP()        ▷ An open list for each weight
4:     $Q \leftarrow Q_1$                     ▷ The standard open list
5:     $C \leftarrow \{\}$                    ▷ The closed set
6:     $\sigma \leftarrow \varnothing$        ▷ The current solution node
7:     $\epsilon \leftarrow \infty$           ▷ The current error bound
8:     ADDTOFRONTIER($n_0$)
9:     while $\epsilon > 0$ and not ISEMPTY($Q$) do
10:        $w \leftarrow$ RANDOM($\mathcal{W}$)
11:        $n \leftarrow$ PEEK($Q_w$)
12:        DELETEFROMFRONTIER($n$)
13:        if $\sigma = \varnothing$ or $f(n) < f(\sigma)$ then
14:            $C \leftarrow C \cup \{n\}$
15:            for all $c \in$ CHILDNODES($n$) do
16:                if $g(n) +$ COST($n, c$) $+ h(c) < f(\sigma)$ then
17:                    if ISGOAL($c$) then
18:                        $f(c) \leftarrow g(c) \leftarrow g(n) +$ COST($n, c$)
19:                        $\sigma \leftarrow c$
20:                    else if $c \in Q \cup C$ then
21:                        if $g(n) +$ COST($n, c$) $< g(c)$ then
22:                            if $c \in Q$ then
23:                                DELETEFROMFRONTIER($c$)
24:                            else
25:                                $C \leftarrow C \setminus \{c\}$
26:                            $g(c) \leftarrow g(n) +$ COST($n, c$)
27:                            ADDTOFRONTIER($c$)
28:                    else
29:                        $g(c) \leftarrow g(n) +$ COST($n, c$)
30:                        ADDTOFRONTIER($c$)
31:        $\epsilon \leftarrow f(\sigma) - f($PEEK($Q$)$)$
32:    return PATHTOGOAL($\sigma$), $\epsilon$

33: procedure ADDTOFRONTIER($n$)
34:    for all $w \in \{1\} \cup \mathcal{W}$ do    ▷ Adds $n$ to each open list
35:        $f_w(n) \leftarrow g(n) + w \cdot h(n)$
36:        INSERT($Q_w, n, f_w(n)$)   ▷ Inserts with priority $f_w(n)$
37: procedure DELETEFROMFRONTIER($n$)
38:    for all $w \in \{1\} \cup \mathcal{W}$ do   ▷ Deletes $n$ from each open list
39:        DELETE($Q_w, n$)
```

Algorithm 1 describes RWA*. An open list Q_w for each weight $w \in \mathcal{W}$, a standard open list Q (which is the same as Q_1), and the closed set C are initialized (Lines 2-5). The current solution node σ and the error bound ϵ are initialized (Line 6-7). The start node is inserted into the open list for each weight with its appropriate f-value using the procedure ADDTOFRONTIER (Line 8). A loop iterates while the error bound is positive and the standard open list is not empty (Line 9). The weight w is adjusted randomly (Line 10). The open node that has the least f_w-value is selected for expansion (Line 11) and removed from all open lists using the procedure DELETEFROMFRONTIER (Line 12) but it is expanded only if it may lead to a better solution than the current solution i.e., on the condition that its standard f-value is less than that of the current solution or if no solution has been found yet (Line 13). If the node is expanded, then it is added to the closed set (Line 14) and a loop iterates through its child nodes (Line 15). If a child node has a standard f-value lower than that of the current solution i.e., if it may lead to a better solution than the current solution, it proceeds to one of three steps (Line 16). If the child node is a goal node, the standard f-value of the child node is updated and the current solution node is set to the child node (Lines 17-19). If the child node is a duplicate of a node already in the open list or in the closed set but if the new g-value of the child node is less than its old g-value, then it is deleted from either the open lists or the closed set and reinserted into the open list for each weight with the appropriate f-value (Lines 20-27). If the child node is not a duplicate, then it is inserted into the open list for each weight with the appropriate f-value (Lines 28-30). The error bound is set to the difference between the cost of the current solution, as it is an upper bound, and the least f-value in the standard open list, as it is a lower bound (Line 31). The final solution is extracted as the path from the root node to the solution node and it is returned along with the error bound (Line 32). Note that Algorithm 1 is also a description of AWA* with weight w as it is same as RWA* given $\mathcal{W} = \{w\}$.

Experiments

We compare the performance of RWA* to AWA* on a range of benchmark problems in a contract setting using a range of static weights that are often used in weighted heuristic search. Both RWA* and AWA* use a set of weights $\mathcal{W} = \{1, 1.5, 2, 3, 4, 5\}$. The benchmark problems below were carefully selected to reflect domains that require different static weights and domains for which higher weights do not necessarily speed up the search (Wilt and Ruml 2012). The parameters of each benchmark problem were selected to avoid trivializing the problem by either not having enough time for any algorithm to generate a solution or having too much time so that every algorithm finds the optimal solution.

Sliding Puzzle (SP) An SP instance has a set of $J = j^2 - 1$ tiles with each tile j labeled from 1 to J in a $j \times j$ grid. The tiles must be moved from an initial configuration to a desired configuration given a *unit* cost $c(j) = 1$ for moving a tile j. The sum of the Manhattan distance from the initial position of each tile to a desired position is used as an admissible and consistent heuristic function. We use the parameter $j = 4$, a setting commonly known as the *15 puzzle*. The difficulty of an instance, as measured by the h-value of the initial configuration, is chosen between 35 and 45 randomly.

Inverse Sliding Puzzle (ISP) An ISP instance is the same as an SP instance except that there is an *inverse* cost $c(j) = 1/j$ for moving a tile j. This means that the sum of the Manhattan distance from the initial position of each tile to the desired position, weighted by the cost for moving each tile, is used as an admissible and consistent heuristic function.

Traveling Salesman Problem (TSP) A TSP instance has a set of J cities that must be visited along an optimal route given a cost between each pair of cities. We use a *sparse* problem where a percentage of the edges have infinite cost. The total cost of a minimum spanning tree across the unvisited cities (with an infinite cost when no tour is feasible) is used as an admissible and consistent heuristic function. The number of cities J is chosen at random between 15 and 25. The sparsity is chosen between 0% and 30% randomly. The distance between each pair of cities is chosen randomly.

City Navigation Problem (CNP) A CNP instance simulates navigating between two locations that might be in different cities (Wilt and Ruml 2012). There is a set of J cities scattered randomly on a $j \times j$ square such that each city is connected by a random tour and to a set of its nearest n_J cities. Each city also contains a set of K locations scattered randomly throughout the city that is a $k \times k$ square such that each location in the city is connected by a random tour and to its nearest n_K locations. The link between a pair of cities costs the Euclidean distance plus an offset α, and the link between a pair of locations within a city costs the Euclidean distance scaled by a random number sampled from $\mathcal{U}(1, \beta)$. The goal is to determine the optimal path from a randomly selected starting location in one city to another randomly selected location, which may be in another city. The Euclidean distance from the current location to the target location is used as an admissible and consistent heuristic. The values of the parameters are chosen such that $J = 150$, $j = 100$, $n_J = 3$, $K = 150$, $k = 1$, $n_K = 3$, $\alpha = 2$, and $\beta = 1.1$ following recent work (Wilt and Ruml 2012).

Experimental Setup Each approach is evaluated using the mean *solution quality* across a set of 500 random instances for every benchmark problem. For each instance, we perform 1 run of AWA* (as it is deterministic) and take the median of 5 runs of RWA* (as it is stochastic). A solution quality $q = 0$ means no solution while a solution quality $q = 1$ means an optimal solution. Formally, for any instance of a problem, solution quality is ideally defined as the ratio, $q = \zeta^*/\zeta$, where ζ^* is the cost of the optimal solution and ζ is the cost of the final solution. However, it is often not feasible to calculate the cost of the optimal solution for complex problems. Like earlier work (Hansen and Zilberstein 2001; Svegliato, Wray, and Zilberstein 2018; Svegliato, Sharma, and Zilberstein 2020), we estimate solution quality as the ratio, $q = h(s_0)/\zeta$, with $h(s_0)$ as the h-value of the initial state s_0 and ζ as the cost of the final solution.

In every trial, each approach must solve an instance of the benchmark problem within the duration of the contract. The duration of the contract for each trial is set to 1.0 sec for SP and ISP, 0.5 sec for TSP, and 0.4 sec for CNP corresponding to roughly 6000, 3000 and 2400 node expansions on our system. However, we let each approach solve an instance of the benchmark problem until a node expansion limit is reached rather than for the duration of a contract as measured by wall time. We prefer a node expansion limit as it is often used to evaluate search algorithms for consistency/reproducibility.

Experimental Results The results of each benchmark problem demonstrate a counterintuitive behavior: RWA* performs as well or better than AWA*. In particular, RWA* (1) computes solutions with a higher quality on average than any static weight, (2) has the highest probability of comput-

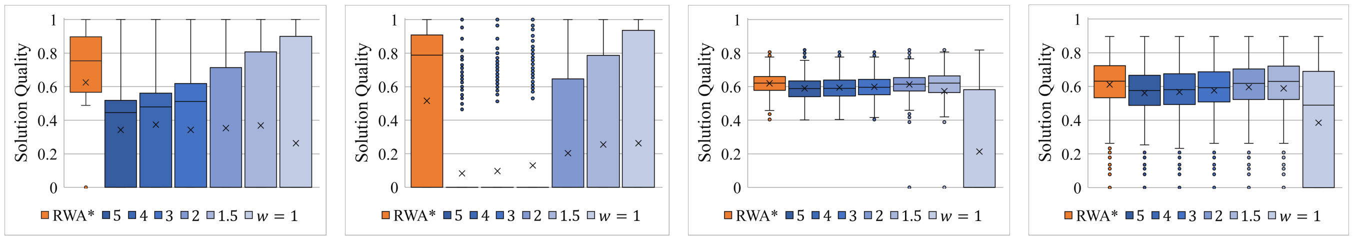

Figure 2: The solution quality box and whisker plots for the SP, ISP, TSP, and CNP benchmark problems (*left* to *right*). The *crosses* denote the mean and the *bullets* denote the outliers. The median and upper quartiles are even zero for some experiments.

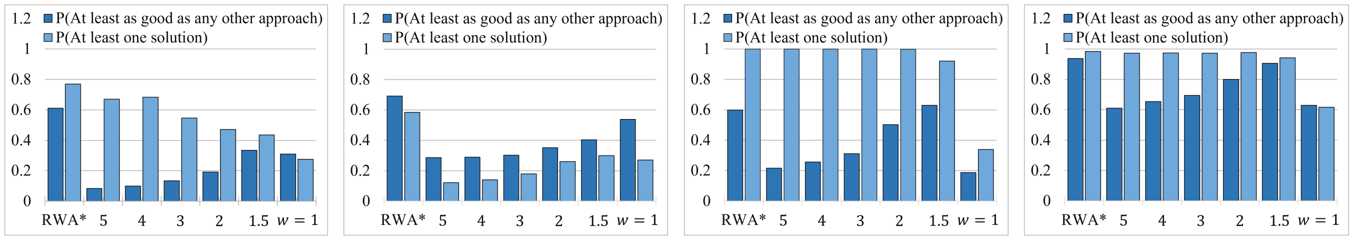

Figure 3: The performance bar graphs for the SP, ISP, TSP, and CNP benchmark problems (*left* to *right*).

ing a solution that is at least as good as any static weight, and (3) has the highest probability of computing at least one solution compared to any static weight.

Fig. 2 shows the qualities for the solutions generated by each approach on random instances of the benchmark problems. For SP, RWA* exhibits a much higher mean solution quality than that of AWA* for all weights. Moreover, as the weight of AWA* decreases from $w = 5$ to $w = 1$, the spread of solution qualities increases because AWA* does not generate any solution to a higher number of instances, but whatever solutions are found tend to be of higher quality. For ISP, RWA* continues to exhibit solution qualities with a substantially higher mean than that of AWA* for all weights. In fact, the mean solution quality for RWA* is almost twice that of AWA*, even when AWA* is at its best weight $w = 1.5$. For TSP and CNP, RWA* exhibits solution qualities with a similar or slightly better mean than AWA* for all weights.

Fig. 3 shows two probabilities for each approach: the probability of computing at least one solution and the probability of computing a solution that is at least as good as any other approach. For SP, TSP, and CNP, the probability of a solution decreases as the weight decreases for AWA*. However, for ISP, the probability of a solution increases as the weight decreases from $w = 5$ to $w = 1.5$ but then the probability of a solution decreases at $w = 1$. This shows that increasing the weight may not always speed up the search (Wilt and Ruml 2012). Intuitively, since RWA* exhibits probabilities similar to or higher than AWA* with any static weight, it is clear that RWA* benefits from using different weights to solve a given instance of a problem.

Fig. 4 shows the advantages of RWA* on a specific instance of the SP benchmark problem. In this example, RWA* finds the best overall solution, generates more solutions, and reduces the upper bound $\bar{q}$ defined by the h-value of the initial state s_0 divided by the min f-value in the open

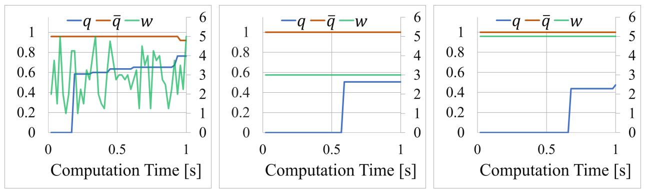

Figure 4: RWA* *(left)* and AWA* with a weight of $w = 3$ and $w = 5$ (*center* and *right*) on a specific instance of the SP benchmark problem. The weight curves are smoothed and plotted on the secondary vertical axis.

list. However, in contrast, AWA* with weights of 1, 1.5, and 2 does not find any solution at all while AWA* with weights of 3 of 5 find a similar solution that is worse than RWA*.

Fig. 5 shows the average solution quality profile of each approach across a range of contracts for each benchmark problem. RWA* performs as well or better than any other approach on average for deadlines greater than a $\sim$500 node-budget ($\sim$0.1s) for SP and ISP, a $\sim$300 node-budget ($\sim$0.05s) for TSP, and a $\sim$150 node-budget ($\sim$0.025s) for CNP. Thus, the advantages of RWA* over AWA* hold across a range of contracts and not just the contracts used earlier.

We make two other important observations that highlight the effectiveness of RWA*. First, even though RWA* is a stochastic algorithm that does not always generate the same solution for a given instance, we observe that the standard deviation of solution quality across individual runs of RWA* on a given instance, on average, is 0.106, 0.045, 0.0009, 0.0002 for SP, ISP, TSP and CNP respectively. This shows that RWA* performs consistently well across different runs on the same instance. Second, the overall results are similar with a finer set of weights $\mathcal{W} = \{1.0, 1.25, 1.5, \ldots, 4.75, 5\}$ for RWA* or with a wall time limit instead of a node expan-

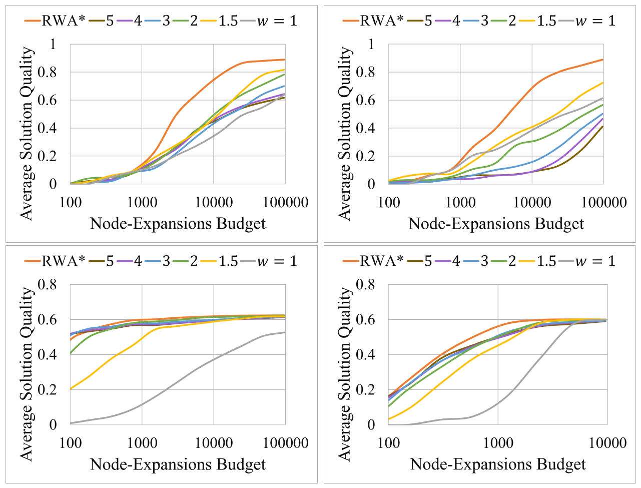

Figure 5: The performance of each approach compared across a range of contract durations on SP, ISP, TSP, and CNP benchmark problems (*top-left* to *bottom-right*).

sion limit when the open lists are operated in parallel.

While we do not provide a definitive explanation for why RWA* outperforms AWA*, our experiments offer some insight. First, we observe that the best static weight can be hard to find as it is specific to an instance, but solution quality tends to increase rapidly once the best static weight is approached. Hence, for most instances, AWA* uses a static weight that is not a good fit for the instance, while RWA* outperforms since it switches between weights randomly and spends at least some time at the ideal weight for the instance. Second, we observe that there are many instances in which RWA* strictly outperforms AWA* even with the best static weight for that instance. This implies that RWA* can compute better solutions by randomly adjusting the weight at runtime for a given instance. In fact, RWA* strictly outperforms AWA* on 30%, 25%, 11%, and 3% of the instances of the SP, ISP, TSP, and CNP benchmark problems.

Our Julia library for AWA*/RWA* is openly available.[1]

Conclusion

We propose RWA*, a randomized variant of AWA*, that randomly adjusts its weight at runtime. On a set of benchmark domains, RWA* typically computes better solutions, exhibits a higher probability of computing any solution at all, and exhibits a higher probability of computing a solution at least as good as any static weight of AWA* in a contract setting across a range of contract durations. Overall, RWA* is an appealing anytime heuristic search algorithm because it is easy to implement and effective without any extensive offline experimentation or parameter tuning.

Acknowledgments

This work was supported in part by the NSF Graduate Research Fellowship DGE-1451512, and NSF grants IIS-1813490 and IIS-1954782.

[1]https://github.com/bhatiaabhinav/AnytimeWeightedAStar.jl

References

Aine, S.; Chakrabarti, P.; and Kumar, R. 2007. AWA*: A window constrained anytime heuristic search algorithm. In *20th International Joint Conference on Artificial Intelligence*, 2250–2255.

Cohen, B.; Chitta, S.; and Likhachev, M. 2014. Single- and dual-arm motion planning with heuristic search. *International Journal of Robotics Research* 33(2): 305–320.

Hansen, E. A.; and Zhou, R. 2007. Anytime heuristic search. *Journal of Artificial Intelligence Research* 28: 267–297.

Hansen, E. A.; and Zilberstein, S. 2001. Monitoring and control of anytime algorithms: A dynamic programming approach. *Artificial Intelligence* 126(1-2): 139–157.

Hansen, E. A.; Zilberstein, S.; and Danilchenko, V. A. 1997. Anytime heuristic search: First results. Technical Report 97-50, Computer Science Department, University of Massachussetts Amherst.

Kobayashi, H.; and Imai, H. 1998. Improvement of the A* algorithm for multiple sequence alignment. *Genome Informatics* 9: 120–130.

Likhachev, M.; Ferguson, D.; Gordon, G.; Stentz, A.; and Thrun, S. 2008. Anytime search in dynamic graphs. *Artificial Intelligence* 172(14): 1613–1643.

Likhachev, M.; Gordon, G. J.; and Thrun, S. 2003. ARA*: Anytime A* with provable bounds on sub-optimality. *Advances in Neural Information Processing Systems* 16: 767–774.

Pohl, I. 1970. Heuristic search viewed as path finding in a graph. *Artificial Intelligence* 1(3-4): 193–204.

Richter, S.; Thayer, J. T.; and Ruml, W. 2010. The joy of forgetting: Faster anytime search via restarting. In *20th International Conference on Automated Planning and Scheduling*, 137–144.

Sun, X.; Druzdzel, M. J.; and Yuan, C. 2007. Dynamic weighting A* search-based MAP algorithm for Bayesian networks. In *20th International Joint Conference on Artificial Intelligence*, 2385–2390.

Svegliato, J.; Sharma, P.; and Zilberstein, S. 2020. A model-free approach to meta-level control of anytime algorithms. In *IEEE International Conference on Robotics and Automation*.

Svegliato, J.; Wray, K. H.; and Zilberstein, S. 2018. Meta-level control of anytime algorithms with online performance prediction. In *27th International Joint Conference on Artificial Intelligence*.

Thayer, J.; and Ruml, W. 2009. Using distance estimates in heuristic search. In *19th International Conference on Automated Planning and Scheduling*, 382–385.

Thayer, J.; and Ruml, W. 2010. Anytime heuristic search: Frameworks and algorithms. In *3rd Annual Symposium on Combinatorial Search*, 121–128.

Thayer, J. T.; and Ruml, W. 2008. Faster than weighted A*: An optimistic approach to bounded suboptimal search. In *18th International Conference on Automated Planning and Scheduling*, 355–362.

Van Den Berg, J.; Shah, R.; Huang, A.; and Goldberg, K. 2011. ANA*: Anytime nonparametric A*. In *25th AAAI Conference on Artificial Intelligence*, 105–111.

Wilt, C.; and Ruml, W. 2012. When does weighted A* fail? In *5th Annual Symposium on Combinatorial Search*, 137–144.

Zhou, R.; and Hansen, E. A. 2002. Multiple sequence alignment using anytime A*. In *18th AAAI Conference on Artificial Intelligence*, 975–977.

Zilberstein, S. 1996. Using anytime algorithms in intelligent systems. *AI Magazine* 17(3): 73–83.