

Anytime Column Search

Satya Gautam Vadlamudi¹, Piyush Gaurav¹, Sandip Aine²,
and Partha Pratim Chakrabarti¹

¹ Indian Institute of Technology Kharagpur, Kharagpur WB 721302, India
`{satya,ppchak}@cse.iitkgp.ernet.in`

² Carnegie Mellon University, Pittsburgh PA 15213, USA
`asandip@andrew.cmu.edu`

Abstract. Anytime heuristic search algorithms are widely applied where best-first search algorithms such as A* require large or often unacceptable amounts of time and memory. Anytime algorithms produce a solution quickly and iteratively improve the solution quality. In this paper, we propose novel anytime heuristic search algorithms with a common underlying strategy called Column Search. The proposed algorithms are complete and guarantee to produce an optimal solution. Experimental results on sliding-tile puzzle problem, traveling salesman problem, and robotic arm trajectory planning problem show the efficacy of proposed methods compared to state-of-the-art anytime heuristic search algorithms.

Keywords: problem solving, heuristic search, anytime algorithms.

1 Introduction

Heuristic search is a generic problem solving technique studied in Artificial Intelligence (AI). A* [8] is the most well-known heuristic search algorithm which has been widely used to solve several problems involving path planning and combinatorial optimization. It explores the state space with the help of two lists: OpenList and ClosedList which contain the nodes to be expanded and the nodes that are already expanded respectively. Initially, OpenList comprises of the start node. Each node has a g -value denoting its distance from the start node and an h -value that estimates its distance to a goal node. The promise of a node n called as f -value equals $g(n) + h(n)$. Most promising node from the OpenList is expanded and added to ClosedList by A* till a goal node is found. A* is an optimal algorithm that expands minimum number of nodes to find an optimal solution, when admissible heuristics are used. However, on large problems, the number of nodes which need to be expanded may be exponential in nature, which makes A* ineffective to use in such cases.

Anytime search algorithms are pursued to address this issue. They quickly generate a solution and then produce improved quality solutions with the progress of time. Several anytime algorithms were proposed based on A*. Anytime Weighted A* [6,7] uses weighted heuristics while evaluating f -values of nodes, which helps in converging to suboptimal solutions quickly, and the search is continued to produce results in an anytime manner. Anytime Restricted A* (ARA*) [11] follows similar concept but decreases weights after each iteration to improve the performance.

Anytime Nonparametric A* (ANA*) [15] improvises on ARA* by analyzing the theory behind the greedy approach followed by ARA* and gets rid of the weight-parameter tuning.

On the other hand, algorithms that do not rely on heuristic manipulation (via weights, like above) such as depth-first branch and bound (DFBB) [10] are also used as anytime algorithms. Beam search [4] is another such simple algorithm that is widely used which gives an approximate solution. It expands beam-width number of most promising nodes at each level (depth of the search tree) until a solution is found or the depth limit of the search tree is reached, where beam-width is taken as the input parameter. However, it is not complete. Beam-stack search [17] is proposed to complete beam search which also gives anytime performance. It uses a novel data structure called beam-stack to keep track of the nodes being expanded and the ones to be expanded by chronological backtracking, and also works within the given beam-width.

Anytime Window A* (AWA*) [2] is another recent anytime algorithm which uses a window based cutoff for the nodes to be present in open list. It ensures completeness by maintaining a suspend list for nodes that are being removed from open list which are considered for expansion (brought back to open list) at a later point in time. Later, Memory-bounded Anytime Window A* (MAWA*) [14] was proposed based on AWA* and Memory-bounded A* (MA*) [5] to be able to work in restricted memory conditions.

In this paper, we propose a novel yet simple anytime heuristic search algorithm called Anytime Column Search (ACS) that is complete, which takes the column-width as parameter. We also explore several non-parametric versions of the proposed algorithm. The basic idea is to explore column-width number of most promising nodes at each level and put them in closed list, and repeat this until the open lists of all levels become empty. The concept is intuitive and augurs well with anytime objectives such as finding initial solution quickly and improving it with the progress of time.

ACS is similar to beam search up-to an extent, however, we choose a different name from ‘beam’ as it is traditionally related (and hence expected) to be working within the given beam-width. While beam search can be highly non-monotonic [1], ACS promises to recover from such effect faced in its first iteration as it continues the search in an admissible manner. Most importantly, it does not involve any re-expansions of the same node, which distinguishes it from algorithms such as naive iterative beam search and complete anytime beam search. For example, complete anytime beam search [16] adopts iterative weakening strategy, *restarting* beam search with weaker pruning rules in each iteration, which results in node re-expansions.

Apart from the anytime performance, ACS also has a unique ability of reporting solutions at regular intervals (due to bounded number of node expansions in each iteration). This feature can also be exploited for using it in a contract search environment [3]. Since it looks at nodes at all levels in each iteration, it can be expected to return best possible solution in a given iteration with the corresponding bound on the number of node expansions.

The rest of the paper is organized as follows: In Section 2, we present the proposed methods which includes Anytime Column Search and its non-parametric variants. In Section 3, we present the experimental results of proposed algorithms obtained on the benchmark problems of three different domains compared with various state-of-the-art anytime heuristic search algorithms. We conclude in Section 4.

2 Proposed Methods

In this section, we present the proposed anytime heuristic search algorithms. Firstly, we present the Anytime Column Search (ACS) algorithm which takes column-width w as parameter. Later, we describe its non-parametric variants.

2.1 Anytime Column Search (ACS)

Anytime Column Search is a simple and complete anytime heuristic search algorithm. It takes column-width w as parameter and expands w most promising nodes of *OpenList* at each level in each iteration and puts them in *ClosedList* until a goal node is found. Visually, the set of nodes expanded in each iteration form a *column* (see Figure 1). One may also look at it as a sliding window moving from left to right.

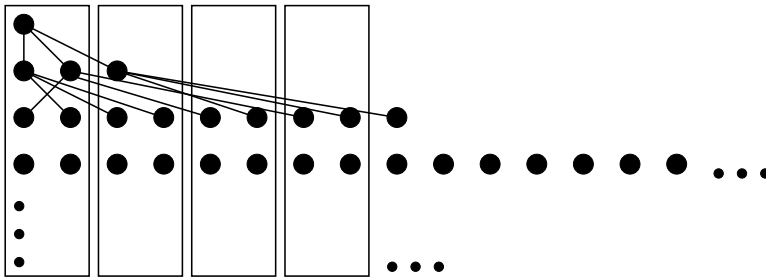


Fig. 1. Progress of ACS₂ on a typical search tree

When using admissible heuristics, the best solution obtained in anytime manner can be used as cutoff as the search progresses (admissible pruning). We denote ACS instance with column-width w as ACS_w . Figure 1 shows the progress of ACS_2 on a typical search tree. Four columns, each of width 2, are shown which are explored from left to right. No edge goes from a given column to any column on its left side on a search *tree* as the right-side columns appear only after the complete exploration of the left-side columns and no node is re-expanded.

ACS₁ is particularly interesting for the following reason: Anytime algorithms are aimed at achieving two main targets, a quick solution, and frequent improved solutions with the progress of time. In this scenario, ACS₁ appears to be a simple base level anytime algorithm which promises both the initial quick solution and frequent solution improvements (even attempted to be regular in time) which

looks at the states of all levels in each iteration (hence appearing to be more informed and calculated while trying to produce the next solution in anytime manner).

Algorithm 1. Anytime Column Search (ACS)

```

1: INPUT :: A search graph  $G$ , a start node  $s$ , and column-width  $w$ .
2:  $BestSol \leftarrow \infty$ ;  $g(s) \leftarrow 0$ ; Calculate  $f(s)$ ;  $Level(s) \leftarrow 0$ ;
    $OpenList(0) \leftarrow \{s\}$ ;  $OpenList(i) \leftarrow \phi, \forall i(0 < i < MAX\_DEPTH)$ ;
    $ClosedList \leftarrow \phi$ ;
3: while  $\exists i OpenList(i) \neq \phi$  do
4:   for  $i = 0$  to  $MAX\_DEPTH - 1$  do
5:     for  $w$  number of times do
6:       if  $OpenList(i) = \phi$  then
7:         break;
8:       end if
9:        $n \leftarrow$  least  $f$ -valued node from  $OpenList(i)$ ; (for minimization problem)
10:      if  $IsGoal(n)$  then
11:        if  $BestSol > f(n)$  then
12:           $BestSol \leftarrow f(n)$ ;  $Goal \leftarrow n$ ;
13:        end if
14:        Move  $n$  from  $OpenList(i)$  to  $ClosedList$ ;
15:        continue;
16:      end if
17:      GenerateChildren( $n$ );
18:      Move  $n$  from  $OpenList(i)$  to  $ClosedList$ ;
19:    end for
20:  end for
21: end while
22: return  $BestSol, Goal$ ;

```

ACS Details: Algorithm 1 presents the proposed method ACS. It maintains different *OpenLists* for nodes at different levels. As long as there exists a non-empty list, it expands w most promising nodes at each level from depth 0 till the maximum allowed depth. It does not assume that the heuristic being used is admissible. When admissible heuristics are being used, one can perform a check after line 9 as to whether the f -value of node being explored is less than the best solution or not. If the f -value is greater than or equal to that of current best solution (in case of a minimization problem), the node can be pruned admissibly, and the search can be moved to next level.

GenerateChildren routine (Algorithm 2) generates all successors of the input node and checks for their existence already in doing so. If a successor exists with a better path from start node already, then it is not disturbed. When using admissible heuristics, f -value of a child can be checked against $BestSol$ before it is inserted into the corresponding *OpenList* and the child node can be pruned admissibly if the f -value exceeds $BestSol$.

Algorithm 2. GenerateChildren

```

1: INPUT :: Node  $n$  whose children are to be generated, and the lists.
2: if  $Level(n) = MAX\_DEPTH - 1$  then
3:   return;
4: end if
5: for each successor  $n'$  of  $n$  do
6:   if  $n'$  is not OpenLists and ClosedLists then
7:      $Level(n') \leftarrow Level(n) + 1$ ;
8:     Insert  $n'$  to OpenList( $Level(n')$ );
9:   else if  $g(n') < \text{its previous } g\text{-value}$  then
10:    Update  $Level(n'), g(n'), f(n')$ ;
11:    Insert  $n'$  to OpenList( $Level(n')$ );
12:   end if
13: end for

```

Theorem 1. *ACS is complete and guarantees terminating with an optimal solution, provided MAX_DEPTH is at least as large as the number of nodes on a minimum-length optimal solution path and the search is not constrained by the memory available.*

Proof. The complete search graph is explored in a column by column manner up-to the maximum depth limit given. Pruning of nodes is only done when using admissible heuristics where the f -values of corresponding nodes are found out to be greater than or equal to that of best solution obtained, which is admissible. Therefore, no potential node exists which is left unexpanded, that can lead to a better solution than that of the final solution. $\square$

Lemma 1. *ACS does not re-expand any node unless a better path has been found from the start node to that node.*

Note that, the node expansions in the first iteration are same as that of beam search with beam-width w . In that sense, the ACS algorithm can be viewed as another way of making beam search anytime and complete.

Another point to note is that, the ACS algorithm takes column-width w as parameter, and its anytime performance will depend on this input parameter. We present the comparison of ACS instances with different column-widths in Section 3. Column-width can be chosen such that the solutions are reported at required intervals (assuming that goal nodes are found as higher depths are explored, an estimate of the intervals at which a better solution can be expected may be calculated for a particular column-width).

In this paper, we also present non-parametric versions of ACS using simple modifications which are explained next.

2.2 Anytime Column Progressive Search (ACPS)

ACS algorithm expands w nodes from *OpenList* at each level in each iteration, where w is taken as input parameter. The idea in Anytime Column Progressive

Search (ACPS) is to progressively increase w from an initial value of 1 by a value of 1 in each iteration. It means that the column-width in i^{th} iteration in case of ACPS is i (in contrast to column-width being w in all iterations in case of ACS). The algorithm is clearly a non-parametric version of ACS.

2.3 Anytime Column Scaling Search (ACSS)

Instead of increasing column-width by 1 in each iteration, one may greedily reset the column-width to 1 whenever a solution is found, in the hope of obtaining a better solution with lower number of node expansions next time. We call the algorithm with this idea as Anytime Column Scaling Search (ACSS). It increases its column-width by 1 in each iteration as long as better solution is not obtained and resets the column-width to 1 whenever a better solution is found. Such a reset technique was also used in [13] on top of Anytime Window A*.

2.4 Anytime Column Adaptive Search (ACAS)

Another way of handling column-width is as follows: Initialize column-width to 1; do not change the column-width as long as better solutions are found with that column-width; when a better solution is not found in an iteration using the column-width, increase it by 1. We call the algorithm with this strategy as Anytime Column Adaptive Search (ACAS). The motivation is that, perhaps it is necessary to expand at-least the current number of nodes to get an improved solution, while also hoping that it is sufficient. When it is found to be not sufficient (better solution is not found), the column-width is increased.

Next, we present the experimental analysis of proposed methods and their comparison with some of the state-of-the-art anytime heuristic search algorithms.

3 Experimental Results

In this section, we present the experimental results comparing the proposed algorithms against several existing anytime algorithms, namely, Depth-first Branch and Bound (DFBB) [10], Beam-Stack search [17], Anytime Window A* (AWA*) [2], Anytime Repairing A* (ARA*) [11], and Anytime Non-parametric A* (ANA*) [15]. All the experiments have been performed on a Dell Precision T7500 Tower Workstation with Intel Xeon 5600 Series at $3.47\text{-GHz} \times 12$ and 192-GB RAM. We display the results in terms of a metric called % *Optimal Closeness*, which is defined as (for a minimization problem, this metric suggests how close the obtained solution is to an optimal solution):

$$\% \text{ Optimal Closeness} = (\text{Optimal sol.} / \text{Obtained sol.}) \times 100.$$

3.1 Sliding-Tile Puzzle Problem

For our experiments, we have considered all 50 24-puzzle instances from Korf et al. [9, Table II] which are referred to with the same index number by which

they were presented, prefixed with K. Manhattan distance heuristic is used as the heuristic estimation function (which underestimates the actual distance to goal). All algorithms are allowed to explore up-to a maximum depth of 1000.

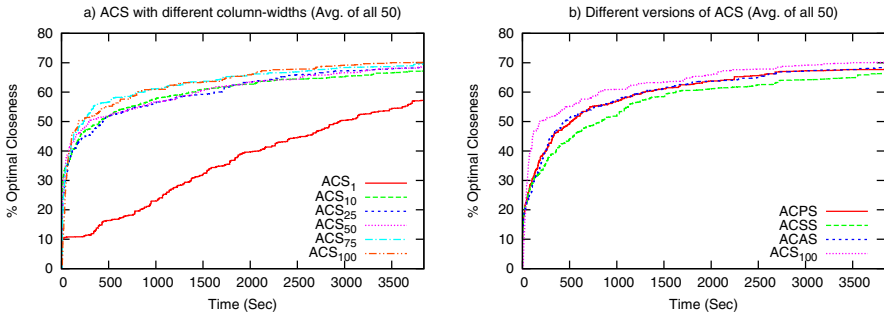


Fig. 2. Comparison of anytime performance of proposed algorithms against each other on the (Sliding-tile) 24-Puzzle instances

Figure 2 presents the comparison of anytime performances of proposed algorithms against each other on the 24-Puzzle instances. This is presented to show the effect of column-width parameter on the performance of ACS as well as to find the best working algorithm for the 24-puzzle domain. From the plots, one can see that ACS₁₀₀ outperforms the other instances. Also note that, performances with column-widths ≥ 10 are very similar which may be due to the low branching factor of the puzzle problem.

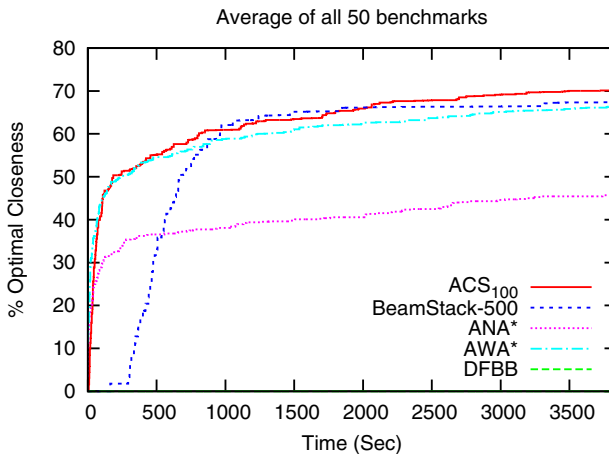


Fig. 3. Comparison of anytime performance of ACS₁₀₀ with some of the existing anytime algorithms on the (Sliding-tile) 24-Puzzle instances. DFBB could not come up with any solution on any of the instances.

Figure 3 presents the comparison of anytime performance of ACS₁₀₀ with some of the state-of-the-art anytime algorithms on the (Sliding-tile) 24-Puzzle instances. At initial stages, ACS outperforms beam stack search and is comparable to AWA* (up-to 1000 Sec.), and at the later stages beam stack search becomes good but ACS remains comparable. Note that, beam stack search is affected by the initial lag due to time taken by beam search with such large beam-width, which is the first step of beam-stack search. When lower beam-widths are used to plug the initial lag of Beam-stack search, it resulted in the degradation of its anytime performance through out the 1 hour window.

Table 1. Comparison of different algorithms on the 50 (Sliding-tile) 24-Puzzle benchmarks

Top Count vs. Time (Mins.)											
Algorithm	1	2	4	8	12	16	24	32	48	64	
ACS ₁₀₀	22	25	22	18	11	9	10	14	19	17	
BeamStack-500	0	0	1	19	30	34	33	29	23	24	
ANA*	7	6	4	3	1	0	0	0	2	1	
AWA*	19	19	23	12	11	10	10	11	12	14	
DFBB	0	0	0	0	0	0	0	0	0	0	

Table 1 presents the comparison of different algorithms on Puzzle problem based on the Top count. Top count indicates the number of times an algorithm has produced the best solution in the corresponding column (time) over all 50 24-Puzzle benchmarks of [9]. This measure gives a complementary picture to the plots showing the average values by showing the number of instances on which an algorithm has been found better than others (at different time instants). Note that, the average value of an algorithm may become high if it outperforms other algorithms significantly on a corner case whereas it may be bad in a number of other cases. Such cases can be detected via *Top Count*. Note that, on any instance, multiple algorithms may produce the best solution in a given time. On two instances, none of the algorithms could produce a solution in 1 Minute within the depth of 1000. Clearly, beam stack search outperforms all the other algorithms in this domain once it starts producing the solutions (after 4 Mins.). However, using lower beam-width values to produce the first solution quickly has degraded its anytime performance over the whole window, in which cases ACS came out on top.

3.2 Traveling Salesman Problem

We chose the first 50 symmetric TSPs (when sorted in increasing order of their sizes) from the traveling salesman problem library (TSPLIB) [12] for our experiments. These range from burma14 to gr202 where the numerical postfixes denote the size of the TSPs. Minimum spanning tree (MST) heuristic is used as the heuristic estimation function (which is an under-estimating heuristic). In the initial state, some city c is chosen as the starting point and in each successive

state the next city n to be visited is chosen (which is not already visited) till all cities are visited.

TSP was often looked as a tree search problem, however, careful examination suggests that if two states denote paths from c to n through a same set of cities S , then only the best of the two need to be pursued further and the other one can be admissibly pruned. This change resulted in tremendous improvement in terms of search state space as well as the anytime performance and the time taken for termination for all heuristic search algorithms. The results obtained with this procedure are presented here. Note that, this may also be looked as a graph search state-space with the state being represented by $\{c, n, S\}$ rather than the traditional way of using the path as the signature of a state (which can just be maintained as an attribute in the current states).

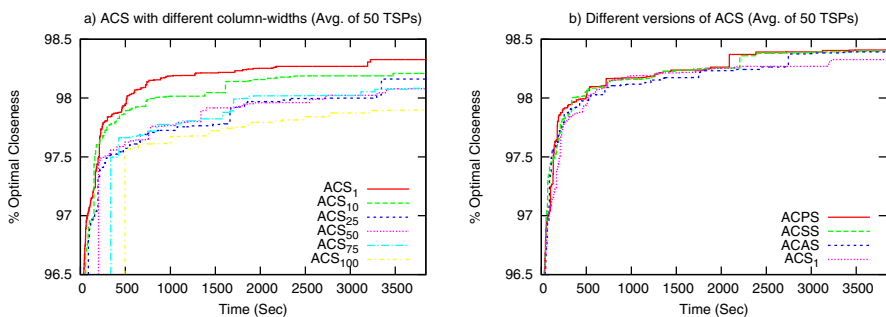


Fig. 4. Comparison of anytime performance of proposed algorithms against each other on various TSP instances

Figure 4 presents the comparison of anytime performance of proposed algorithms against each other on various TSP instances. This is presented to show the effect of column-width parameter on the performance of ACS as well as to find the best working algorithm for the TSP domain. It is interesting to note that ACS₁ outperforms the instances with greater column-widths which shows that in certain cases it is indeed sufficient to expand one column at a time to achieve good anytime performance. From the plots, one can see that all the algorithms are quite effective, and hence we choose ACPS which is the simplest non-parametric algorithm for comparison with the other algorithms.

Figure 5 presents the comparison of anytime performance of ACPS with some of the state-of-the-art anytime algorithms on the TSP instances. Note that, its performance is superior than that of the other algorithms. Only AWA* and Beam-stack search give comparable performances of which Beam-stack algorithm suffers from the initial lag which is explained in Section 3.1. Note that, using larger beam-width for beam-stack algorithm resulted in unacceptable amount of lag at the beginning (delay in producing its first solution).

Table 2 presents the comparison of different algorithms on TSP instances based on the Top count. Top count (whose significance is explained in Section 3.1) indicates the number of times an algorithm has produced the best solution in

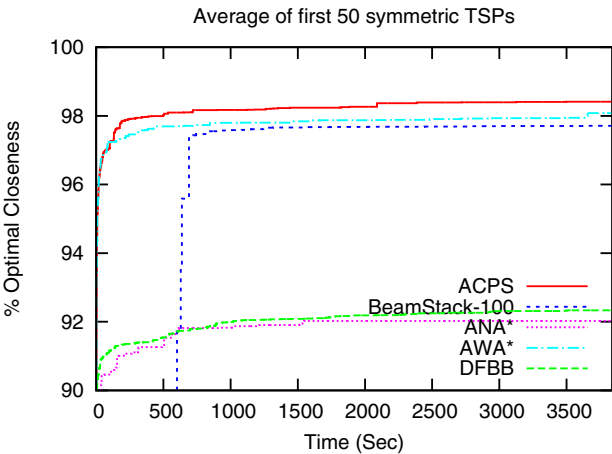


Fig. 5. Comparison of anytime performance of ACPS and the existing search algorithms on various TSP instances

Table 2. Comparison of different algorithms on the 50 TSP benchmarks

Top Count vs. Time (Mins.)											
Algorithm	1	2	4	8	12	16	24	32	48	64	
ACPS	35	32	35	36	40	41	41	41	42	42	
BeamStack-100	21	22	26	26	26	27	28	27	27	27	
ANA*	13	11	13	13	12	12	12	13	13	13	
AWA*	28	32	27	29	29	28	27	27	31	30	
DFBB	8	10	10	11	11	11	11	11	11	12	

the corresponding column (time) over the 50 TSP benchmarks considered. Note that, on any instance, multiple algorithms may produce the best solution in a given time. Performance of ACPS is clearly better than that of all the other algorithms.

3.3 Robotic-Arm Trajectory Planning

We consider both the 6-degree-of-freedom (DOF) arm instances and the 20-DOF arm instances which are available in Likhachev’s software library¹, with fixed base in a 2D environment with obstacles. The objective is to move the end-effector from its initial location to a goal location while avoiding obstacles. An action is defined as a change of a global angle of any particular joint. Cost of each action is taken as one. The environment is discretized into a 50x50 2D grid. Joint angles are discretized so as to never move the end-effector by more than one cell on the 50x50 grid in a single action. The heuristic is calculated as the shortest distance from the current location of the end-effector to the goal location that avoids obstacles (which is an underestimating heuristic).

¹ <http://www.cs.cmu.edu/~maxim/software.html>

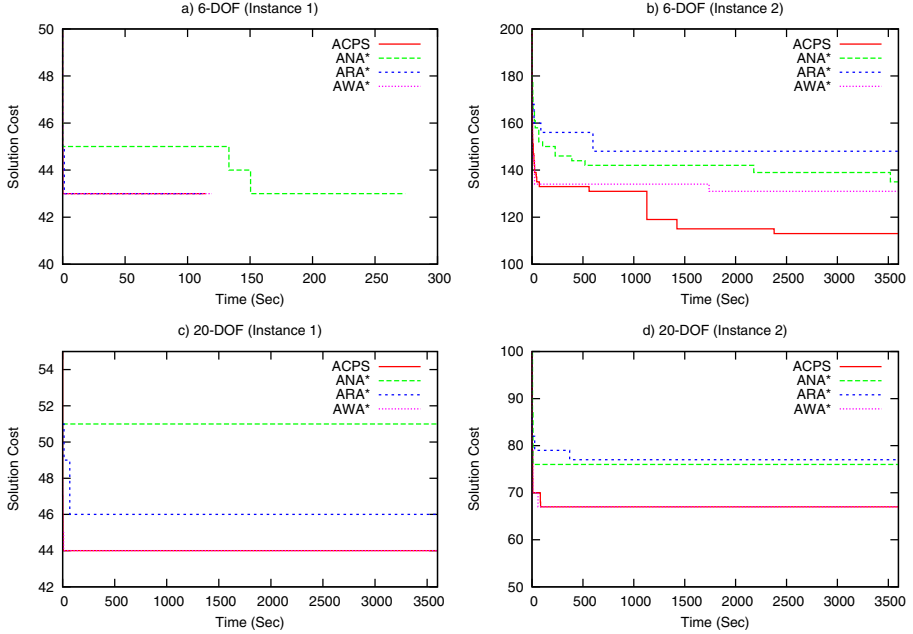


Fig. 6. Comparison of anytime performance of ACPS and the existing search algorithms on various robotic-arm trajectory planning benchmarks

We compared our algorithm ACPS (which gave the best performance in ACS family methods) with the anytime algorithms: ARA*, ANA*, and AWA*, of which ARA* and ANA* implementations came with the software to which we added our implementations of ACS family algorithms and AWA* for comparison. Figure 6 shows the results obtained on the four robotic-arm instances considered (a total of only 5 benchmarks were available out of which the results of a 6-DOF instance are not presented here due to space constraints, which were observed to be similar). Here, the actual cost of the solution is used in the plots instead of *% Optimal Closeness* as the optimal solutions are not available for all instances to compute the closeness. The parameters of ARA*: ϵ_0 and $\Delta\epsilon$ are tuned for best performance. $\Delta\epsilon$ is set to 0.2 as recommended in [11] and ϵ_0 is tested with various values $\in [2, 100]$, out of which the best performance was found at 5, which is used in the plots. Clearly, the performance of ACPS is either comparable or better than that of the other algorithms in this case as well.

4 Conclusion

In this paper, we present a simple and efficient anytime heuristic search algorithm called Anytime Column Search (ACS). ACS takes column-width as parameter for its working and guarantees to find optimal solution. Non-parametric versions of ACS are also presented which also demonstrates how it can be easily tuned

as per the working domain. Experimental results on Sliding-tile puzzle problem, Traveling salesman problem (TSP), and Robotic arm trajectory planning demonstrate that the anytime performance of the proposed methods is either superior or comparable to that of state-of-the-art anytime heuristic search algorithms.

Acknowledgement. S. G. Vadlamudi thanks SAP Labs India for their support through Doctoral Fellowship.

References

1. Aine, S., Chakrabarti, P.P.: An analysis of breadth-first beam search using uniform cost trees. In: The Eleventh International Symposium on Artificial Intelligence and Mathematics (January 2010)
2. Aine, S., Chakrabarti, P.P., Kumar, R.: AWA* - A window constrained anytime heuristic search algorithm. In: Veloso, M.M. (ed.) IJCAI, pp. 2250–2255 (2007)
3. Aine, S., Chakrabarti, P.P., Kumar, R.: Heuristic search under contract. *Computational Intelligence* 26(4), 386–419 (2010)
4. Bisiani, R.: Beam search. In: *Encyclopedia of Artificial Intelligence*, pp. 56–58 (1987)
5. Chakrabarti, P.P., Ghose, S., Acharya, A., Sarkar, S.C.D.: Heuristic search in restricted memory. *Artificial Intelligence* 41(2), 197–221 (1989)
6. Hansen, E.A., Zhou, R.: Anytime heuristic search. *J. Artif. Int. Res.* 28(1), 267–297 (2007)
7. Hansen, E.A., Zilberstein, S., Danilchenko, V.A.: Anytime heuristic search: First results. Technical Report 50, Univ. of Massachusetts (1997)
8. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2), 100–107 (1968)
9. Korf, R.E., Felner, A.: Disjoint pattern database heuristics. *Artif. Intell.* 134(1–2), 9–22 (2002)
10. Lawler, E.L., Wood, D.E.: Branch-and-bound methods: A survey. *Operational Research* 14(4), 699–719 (1966)
11. Likhachev, M., Gordon, G.J., Thrun, S.: ARA*: Anytime A* with provable bounds on sub-optimality. In: *Advances in Neural Information Processing Systems*, vol. 16. MIT Press, Cambridge (2004)
12. Reinelt, G.: TSPLIB - A traveling salesman problem library. *ORSA Journal on Computing* 3, 376–384 (1991)
13. Thayer, J.T., Ruml, W.: Anytime heuristic search: Frameworks and algorithms. In: *SOCS* (2010)
14. Vadlamudi, S.G., Aine, S., Chakrabarti, P.P.: MAWA*—A memory-bounded anytime heuristic-search algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 41(3), 725–735 (2011)
15. van den Berg, J., Shah, R., Huang, A., Goldberg, K.Y.: Anytime nonparametric A*. In: *AAAI* (2011)
16. Zhang, W.: Complete anytime beam search. In: *Proceedings of 14th National Conference of Artificial Intelligence AAAI 1998*, pp. 425–430. AAAI Press (1998)
17. Zhou, R., Hansen, E.A.: Beam-stack search: Integrating backtracking with beam search. In: *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS 2005)*, Monterey, CA, pp. 90–98 (2005)