Anytime Contract Search

Sunandita Patra, Satya Gautam Vadlamudi and Partha Pratim Chakrabarti

Abstract Heuristic search is a fundamental problem solving paradigm in artificial intelligence. We address the problem of developing heuristic search algorithms where intermediate results are sought at intervals of time which may or may not be known apriori. In this paper, we propose an efficient anytime algorithm called Anytime Contract Search (based on the contract search framework) which incrementally explores the state-space with the given contracts (intervals of reporting). The algorithm works without restarting and dynamically adapts for the next iteration based on the current contract and the currently explored state-space. The proposed method is complete on bounded graphs. Experimental results with different contract sequences on the Sliding-tile Puzzle Problem and the Travelling Salesperson Problem (TSP) show that Anytime Contract Search outperforms some of the state-of-the art anytime search algorithms that are oblivious to the given contracts. Also, the non-parametric version of the proposed algorithm which is oblivious of the reporting intervals (making it an anytime algorithm) performs well compared to many available schemes.

1 Introduction

Heuristic search has been widely applied over the years in diverse domains involving planning and combinatorial optimization [15]. It is one of the fundamental problem solving techniques of artificial intelligence [14]. A* [6] is the central algorithm around which most other state-of-the-art methods are developed. Owing to the large

S. Patra (✉) · S. G. Vadlamudi · P. P. Chakrabarti
Indian Institute of Technology Kharagpur, Kharagpur, West Bengal 721302, India
e-mail: sunandita.patra@cse.iitkgp.ernet.in

S. G. Vadlamudi
e-mail: satya@cse.iitkgp.ernet.in

P. P. Chakrabarti
e-mail: ppchak@cse.iitkgp.ernet.in

M. Bramer and M. Petridis (eds.), *Research and Development in Intelligent Systems XXX*,
DOI: 10.1007/978-3-319-02621-3_10,

amount of time required by A* algorithm to produce a solution (which is guaranteed to be optimal) in case of complex problems, several themes are pursued that can report solutions (possibly sub-optimal) quickly. Most prominent amongst them is the class of anytime search algorithms [4]. The objective of anytime search algorithms is to produce a solution quickly and improve upon it as time passes. Several methods are proposed in the literature to address this problem.

In this paper, we explore a new dimension to this problem, where a user applying the anytime algorithms typically checks the progress of the algorithm (for any improved solutions) at certain periods of time rather than continuously monitoring it. Or equivalently, it can be said that he/she expects the anytime algorithm to improve upon its current solution after a certain amount of time. We propose to formulate this behavior formally by taking as input a series of timepoints from the user at which he/she would like to get an update from the anytime algorithm.We call this problem, the *anytime contract search problem*, which takes in a Contract series (a series of timepoints) as input at which improved solutions are sought.Equivalently, one may feed the algorithm during its run by the next contract (at which a better solution is sought), instead of giving the whole series apriori. We solve this problem by intelligently combining the strategies of the anytime algorithms and the contract search algorithms.

The basic contract search problem aims at finding the best possible solution in the given time. In [7], a time constrained search algorithm is proposed for solving this problem based on Weighted A* [10]. Contract search algorithm [2, 3] uses probabilistic models to distribute the contract as node expansions across different levels. Deadline aware search algorithm [5] proceeds in a best-first manner, but ensuring that those nodes are chosen for expansion which can lead to a goal state within the given time.

In this work, we follow the approach of distributing the contract as node expansions at different levels to reach a solution within the given time. We learn dynamically to come up with a chosen distribution of contracts and the algorithm progresses without any restarts ensuring no unnecessary re-expansion of nodes. The key contributions of this work are as follows:

1. The introduction of the anytime contract search problem,
2. An efficient algorithm called Anytime Contract Search (ACTR) to solve this problem, and a modification of the algorithm called Oblivious Anytime Contract Search (OACTR) that makes it usable as a traditional anytime algorithm, and
3. Experimental results comparing the proposed methods with some of the state-of-the-art algorithms such as AWA*, ANA*, etc. on the Sliding-tile Puzzle Problem and the Traveling Salesperson Problem where the proposed algorithms are observed to be outperforming the other methods significantly. Further, the performance of the proposed algorithm is analyzed with respect to different input Contract series distributions.

The rest of the paper is organized as follows: In Sect. 2, we present the proposed methods, namely, anytime contract search algorithm and the oblivious version of it along with some of the important properties satisfied by the algorithms. In Sect. 3,

the implementation details and comparison of the proposed methods with some of the state-of-the-art anytime algorithms are presented on the Sliding-tile Puzzle problem and the Traveling Salesperson Problem. Finally, we present a brief discussion pointing to the scope for improvements and future research directions, and conclude, in Sect. 4.

2 Proposed Methods

In this section, we present the Anytime Contract Search algorithm (ACTR) that takes in a series of timepoints at which the user would like to note the progress, and tries to report the best possible solutions at those timepoints.Note that, giving the entire series of timepoints apriori is not mandatory and one may instead dynamically give the contract inputs in a step by step manner during the run of the algorithm after observing the result of the algorithm in the current iteration, as discussed in the Introduction.

The method works by distributing the contract available (till the next report time) into number of nodes to be expanded at each level of the search graph, similar to the idea of Contract Search [2, 3]. However, instead of using probabilistic rank profiles for distributing the contract, our method attempts to learn and adapt the contract distribution scheme based on the feedback from its own previous iterations.Note that, the basic contract search algorithms [2, 3] are not anytime in nature and hence can not be compared with the methods in this work. The previous methods do not have concept of producing multiple solutions but aim for only one best possible solution within the given time.While they use probabilistic rank profiles for their operations, one may either use such approach during the first iteration of the methods proposed in this work, or alternatively, use a simpler beam-search like equal distribution of contracts for different levels.Here, our main focus is not on the initial distribution, rather the later dynamic learning and solving the anytime contract search problem. However, all aspects are open for exploration as discussed in Sect. 4.

Algorithm 1 presents the proposed approach (`ACTR`). It takes as input the search graph, start node, the sequence of timepoints (or just the next timepoint) at which solutions are sought, and also a parameter defining the maximum limit of contract that can be distributed in a given iteration. The last input was induced since we observed that using large contracts directly hampers the learning process of distribution scheme resulting in a poor performance. Separate open lists are maintained for storing the nodes of different levels which are to be expanded, which helps in choosing the most promising node across different levels whose contract limit has not been reached. $ExpCount$ array keeps track of the number of nodes expanded at different levels. $ExpLimit$ array contains the information of maximum number of nodes that can be expanded at different levels as decided by the `DistributeContract` routine. Initially, the $ExpLimit$ value for each level is set to 1 so that the `DistributeContract` routine can allocate equal limits for all levels.

Algorithm 1 Anytime Contract Search (ACTR)

1: **INPUT ::** A search graph, a start node s, maximum contract per iteration, and the Contracts series C of length n.
2: $BestSol \leftarrow infinity$; $g(s) \leftarrow 0$; Calculate $f(s)$; $Level(s) \leftarrow 0$; $OpenList(0) \leftarrow \{s\}$; $ClosedList \leftarrow \phi$; $OpenList(i) \leftarrow \phi$; $\forall i (0 < i < MAX\_DEPTH)$;
3: $ExpCount(i) \leftarrow 0$, $ExpLimit(i) \leftarrow 1$, $\forall i (0 \leq i < MAX\_DEPTH)$;
4: **for** $i \leftarrow 1$ to n and $\exists j\ OpenList(j) \neq \phi$ **do**
5: **while** $C(i) - time\_elapsed > max\_ctr\_per\_iter$ and $\exists j\ OpenList(j) \neq \phi$ **do**
6: $ExpLimit \leftarrow$**DistributeContract**$(max\_ctr\_per\_iter, ExpLimit, ExpCount)$;
7: $BestSol \leftarrow$ **SearchForSolution**$(ExpLimit, BestSol, OpenList, ClosedList, ExpCount)$;
8: $ExpLimit \leftarrow$**DistributeContract**$(C(i) - time\_elapsed, ExpLimit, ExpCount)$;
9: $BestSol \leftarrow$ **SearchForSolution**$(ExpLimit, BestSol, OpenList, ClosedList, ExpCount)$;
10: return $BestSol$;

After the initialization (Lines 2 and 3), the method invokes `DistributeContract` routine and `SearchForSolution` routine in tandem according to the given Contracts series and the time-elapsed to come up with the best possible solutions. The *for* loop from Lines 4–9 indicates execution for each of the contracts, as long as there exists at-least one level at which the $OpenList$ is not empty (since all levels being empty would mean that the search is complete and an optimal solution is found, if exists). As mentioned before, for better learning and better performance of the algorithm, when the contract to be distributed is larger than a pre-defined limit per iteration, it is sub-divided into several blocks of the size of the maximum limit (Lines 5–7) followed by the remaining time (Lines 8–9). For example, if the total contract for the current iteration is given to be 10 min and the maximum contract limit per invocation of `SearchForSolution` routine is set to 4 min, then the while loop (Lines 5–7) executes two times with the maximum limit– 4 min, and then Lines 8–9 execute with the remaining time 2 min.

Algorithm 2 DistributeContract

1: **INPUT ::** Contract to be distributed c, $ExpLimit$, $ExpCount$, and a pre-defined tunable parameter $\alpha \in (0, 1)$.
2: **for** $i \leftarrow 0$ to $MAX\_DEPTH - 1$ **do**
3: $ExpRatio(i) \leftarrow \alpha \times ExpLimit(i) + (1 - \alpha) \times ExpCount(i)$;
4: **for** $i \leftarrow 0$ to $MAX\_DEPTH - 1$ **do**
5: $ExpRatio(i) \leftarrow ExpRatio(i) / \Sigma_j ExpRatio(j)$;
6: $ExpLimit(i) \leftarrow ExpLimit(i) + ExpRatio(i) \times c \times node\_expansion\_rate$;
7: return $ExpLimit$;

The `DistributeContract` routine takes as input: the contract to be distributed, the current $ExpLimit$ and $ExpCount$ of various levels, and a pre-defined tunable parameter $\alpha \in (0, 1)$. For deciding the node expansion ratios of different levels, it takes into account the previous node expansion limit assigned and the actual

number of node expansions that happened at different levels. While the former represents the promise of nodes at different levels as assessed previously, the latter acts as feedback as to whether the nodes of that level turned out to be globally competitive. Clearly, this is one of the many possible schemes that can be used here, more discussion on which is provided later. Different values of α can be tested with to find the most suited one for the given domain. Lines 2–3 of the `DistributeContract` routine show the decision on the expansion ratios of different levels, and Lines 4–6 show the normalization of the ratios and updating of the node expansion limits based on the given contract and the node expansion rate. Here, note that, $ExpRatio(i)$ is a real number (between 0 and 1), while $ExpLimit(i)$ has to be an integer which is guaranteed by using the *math floor* function while computing the same. Node expansion rate (number of nodes expanded per second (unit of time)) is an user input/parameter which is pre-computed for each domain and a given problem size. One may explore using more complex profiling of the node expansion rates if it varies for nodes belonging to different levels, and adjust the contract distribution accordingly.

Algorithm 3 SearchForSolution

1: **INPUT ::** $ExpLimit$, $BestSol$, $OpenLists$, $ClosedList$, and $ExpCounts$.
2: **while** $\exists i$ such that $OpenList(i) \neq \phi$ and $ExpCount(i) < ExpLimit(i)$ **do**
3: $n \leftarrow$ least f-valued node from all the $OpenLists$ at different levels (i) for which $ExpCount(i) < ExpLimit(i)$;
4: **if** IsGoal(n) **then**
5: **if** $BestSol > f(n)$ **then**
6: $BestSol \leftarrow f(n)$;
7: Move n from its $OpenList$ to $ClosedList$; **continue**;
8: **GenerateChildren(**n**)**; Move n from its $OpenList$ to $ClosedList$;
9: $ExpCount(Level(n)) \leftarrow ExpCount(Level(n)) + 1$;
10: return $BestSol$;

The `SearchForSolution` routine takes the node expansion limits, the current best solution, the lists and the node expansion counts as input and searches for a better solution. It chooses the most promising node from lists of all levels whose node expansion limit has not been reached. The node is checked as to whether it is a goal node, and if it is, the current best solution is updated, otherwise, its children are generated and the corresponding expansion count is updated. The process is continued until either all the $OpenLists$ become empty or the node expansion limits for all levels are reached. Note that when using admissible heuristics, one could terminate this routine when the f-value of the node chosen is greater than or equal to that of the current best solution (for a minimization problem), after pruning such nodes.

Lastly, the `GenerateChildren` routine takes in the node to be expanded and the lists, and generates the children of the node. The children are checked as to whether they are already present in the memory and if so they are updated with the currently known best path from the start node. The lists are updated accordingly as per the new levels of the children.

Algorithm 4 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: **for** each successor n' of n **do**
5: **if** n' is not $OpenLists$ and $ClosedLists$ **then**
6: $Level(n') \leftarrow Level(n) + 1$; Insert n' to $OpenList(Level(n'))$;
7: **else if** $g(n') <$ its previous g-value **then**
8: Update $Level(n')$, $g(n')$, $f(n')$; Insert n' to $OpenList(Level(n'))$;

Next, we present a theorem showing the completeness of the proposed method.

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

It is easy to observe that the theorem holds true since the algorithm does not discard any promising node within MAX_DEPTH when given enough time.

Note that, here, the algorithm guarantees to find an optimal solution (if exists), even when inadmissible heuristics are used. This is because the algorithm does not prune nodes whose f-values are greater than or equal to that of the best solution, hence, covering the entire search space. However, when using admissible heuristics, the user can prune the nodes with f-values greater than or equal to that of the best solution whenever encountered, and therefore leverage the benefit by reducing the search space to be explored.

Another property satisfied by the algorithm which helps in making it efficient is: ACTR does not re-expand any node unless a better path has been found from the start node to that node.

In the following, we present a version of the proposed algorithm called Oblivious Anytime Contract Search (OACTR) which is oblivious to the input Contract series, and hence acts as a simple anytime algorithm.

Oblivious Anytime Contract Search (OACTR)

Some users may be interested in a simple traditional anytime algorithm which does not ask for Contract series as input, and which is expected to improve upon the solutions as soon as it can.

Algorithm 5 Oblivious Anytime Contract Search (OACTR)

1: **INPUT ::** A search graph, and a start node s.
2: $BestSol \leftarrow infinity$; $g(s) \leftarrow 0$; Calculate $f(s)$; $Level(s) \leftarrow 0$; $OpenList(0) \leftarrow \{s\}$; $ClosedList \leftarrow \phi$; $OpenList(i) \leftarrow \phi$; $\forall i (0 < i < MAX\_DEPTH)$;
3: $ExpCount(i) \leftarrow 0$, $ExpLimit(i) \leftarrow 1$, $\forall i (0 \leq i < MAX\_DEPTH)$;
4: **while** $\exists i\ OpenList(i) \neq \phi$ **do**
5: $ExpLimit \leftarrow$**DistributeContract**$(MINIMAL\_C, ExpLimit, ExpCount)$;
6: $BestSol \leftarrow$ **SearchForSolution**$(ExpLimit, BestSol, OpenList, ClosedList, ExpCount)$;
7: return $BestSol$;

Algorithm 5 shows a simplified version of Anytime Contract Search which can suit to such requirement. Here, in each iteration, a better solution is sought by using a certain pre-defined minimal contract $MINIMAL\_C$ chosen as per the problem domain. We call this algorithm Oblivious Anytime Contract Search (OACTR) as it just ignores any Contract series inputs. One may also explore other patterns of varying the value of "$MINIMAL\_C$" dynamically for maximizing the performance.

3 Experimental Results

In this section, we present the experimental results comparing the proposed algorithms against several existing anytime algorithms, namely, Beam-Stack search (BS) [17], Anytime Window A* (AWA*) [1], Iterative Widening (IW) [11, 12], Anytime Non-parametric A* (ANA*) [16], and Depth-First Branch and Bound (DFBB) [9]. All the experiments have been performed on a Dell Precision T7500 Tower Workstation with Intel Xeon 5600 Series at 3.47−GHz × 12 and 192-GB RAM.

Anytime performances of different algorithms are usually compared by plotting the average solution costs of all the testcases, obtained at different timepoints. We too use such strategy in our work where we use a metric called $\%Optimal\ Closeness$ which is defined as: $Optimal\ Solution \times 100/Obtained\ Solution$.

It indicates how close the obtained solution is to an optimal solution for a minimization problem. This helps in normalizing the solution costs of different testcases before the average is taken. However, the average value of the quality of the output of an algorithm may become high if it outperforms other algorithms significantly on few corner cases whereas it may be bad in a number of other cases. Such cases can be detected via `Top Count`.

`Top Count` at time t is defined as the number of instances on which a particular algorithm has produced the best solution cost by that time. For example, let two algorithms A_1 and A_2 be compared on 5 testcases at a given time t. Let us assume that A_1 reports better solutions than A_2 in 3 cases, and A_2 finds the better solution in 1 case, and both A_1 and A_2 come up with the same solution on the remaining testcase. Then, the Top Counts for the given algorithms A_1 and A_2 become 4 and 2 respectively. This indicates that a given algorithm is dominating in so many number of cases at that particular time. Note that, on any instance, multiple algorithms may produce the best solution in a given time, and so, the sum of the top counts of various algorithms at a given time can be greater than the total number of testcases, as shown in the above example. This measure gives a complementary picture to the traditional comparison of the average anytime performances.

In each problem domain, we first compare the proposed algorithm with the existing anytime algorithms at different timepoints of the Contract series given by user using Top Count measure. Note that, while the proposed algorithm is run afresh in each case as per the Contract series, other algorithms are run only once as they are oblivious to the given contract series. Next, we show the comparison of Oblivious Anytime Contract search algorithm with the existing ones. Next, results showing the average

anytime performances of all the algorithms using the *% Optimal Closeness* measure are presented. Finally, we show a comparative analysis of the proposed algorithms run with different Contract series inputs amongst themselves to test their adaptiveness towards the fed inputs.

We used a constant value of $\alpha = 0.5$ (the learning parameter in contract distributions) in all our experiments since it was observed to be the most suitable with the considered setting. And, the maximum contract per iteration is set to 8 min in all the experiments.

3.1 Sliding-Tile Puzzle Problem

We have chosen the 50 24-puzzle instances from [8, Table II] for our experiments. Manhattan distance heuristic is used as the heuristic estimation function (which underestimates the actual distance to goal). All algorithms explore up-to a maximum depth of 1000 levels.

Firstly, we run ACTR with the Contract series: {4, 12, 20, 28, 36, 44, 52, 60} which is an arithmetic progression (AP) having uniformly distributed intervals. We compare the obtained results with that of the existing algorithms such as AWA*, Beam-Stack search (BS), Iterative Beam search (IW), ANA*, and DFBB which are oblivious to the given Contract series. Table 1 shows the results comparing the top counts at the timepoints of the AP Contract series. BS_{500} indicates the results of Beam-stack search algorithm when run with beam-width = 500. The beam-width value is chosen after experimenting with several values such that neither the initial solution of the algorithm is delayed (due to large beam-width) nor the anytime performance is hampered (due to small beam-width). It can be observed that the proposed algorithm dominates the other algorithms at the given timepoints. Note that, here the Top Count value corresponding to ACTR decreases for the last two time contracts. This is because one of the other algorithms must have come up with a better solution than ACTR on one of the testcases by that time. Equivalently, ACTR might have come up with a better solution on one of the testcases and the other algorithm(s) might have bettered it on two other testcases, etc. The point to note is that, Top Count value for an algorithm need not be monotonically increasing, and is dependent on the relative performance of the algorithms at the time point under consideration.

Next, we run ACTR with other Contract series distributions such as a Geometric Progression (GP) and a Randomly generated series (RD). Tables 2 and 3 show the comparison of the obtained results in terms of top count with respect to that of the existing algorithms. Note that, while ACTR is run afresh with the given Contract series, the other algorithms being oblivious to the given Contract series are not effected by it. Only the analysis of Top Counts is carried out with respect to a different set of timepoints in each case. It can be seen that the proposed algorithm outperforms the others in these cases as well. Note that, it is expected that the first columns (corresponding to 4 min.) of Tables 1 and 3 should match, however, the minor discrepancy may be due to ACTR producing a better solution than others

Table 1 Comparison of ACTR with the existing algorithms on the 50 (sliding-tile) 24-puzzle benchmarks when the contract series input is an AP series

	Top count versus **time** (min)							
Algorithm	**4**	**12**	**20**	**28**	**36**	**44**	**52**	**60**
ACTR	**48**	**48**	**49**	**49**	**49**	**49**	**48**	**47**
AWA*	3	4	4	4	4	3	3	4
BS_{500}	1	0	0	0	0	0	0	0
IW	0	0	0	0	0	1	1	1
ANA*	0	0	0	0	0	0	0	0
DFBB	0	0	0	0	0	0	0	0

Table 2 Comparison of ACTR with the existing algorithms on the 50 (sliding-tile) 24-puzzle benchmarks when the contract series input is a GP series

	Top count versus **time** (min)						
Algorithm	**1**	**2**	**4**	**8**	**16**	**32**	**64**
ACTR	**47**	**48**	**47**	**48**	**48**	**46**	**45**
AWA*	6	3	5	3	4	7	7
BS_{500}	2	0	0	1	1	1	1
IW	1	0	0	1	1	1	1
ANA*	0	0	0	0	0	0	0
DFBB	0	0	0	0	0	0	0

Table 3 Comparison of ACTR with the existing algorithms on the 50 (sliding-tile) 24-puzzle benchmarks when the contract series input is generated randomly

	Top count versus **time** (min)							
Algorithm	**4**	**10**	**23**	**26**	**31**	**49**	**54**	**60**
ACTR	**49**	**49**	**48**	**48**	**48**	**48**	**47**	**46**
AWA*	3	3	4	4	3	5	5	6
BS_{500}	1	0	0	0	0	1	0	0
IW	0	0	0	0	0	1	1	1
ANA*	0	0	0	0	0	0	0	0
DFBB	0	0	0	0	0	0	0	0

slightly before the 4 min time limit during the Random-series input execution, and producing a better solution than others slightly after the 4 min time limit during the AP-series input execution. They are expected to perfectly match if the measure is in terms of node expansions, instead of time (which may get effected slightly during different runs of the same algorithm).

Clearly, in the previous cases, the proposed algorithm is expected to gain advantage over the other algorithms as it mends itself as per the given Contract series. Now, we present the comparison of the Oblivious ACTR (OACTR) which is similar to traditional anytime algorithms that do not take any Contract series as input.

Here, OACTR takes as input a minimal contract to be used in each iteration which is set to 1000 node expansions. Table 4 shows the comparative results in terms of top count at uniformly distributed time intervals. The outstanding performance of OACTR suggests that the basic underlying anytime framework proposed is of significance.

Table 5 shows the comparison of average of the anytime performances of the algorithms in terms of $\%\,Optimal\,Closeness$. ACTR_{AP} and ACTR_{GP} denote the runs of ACTR algorithm with the input Contract series being the AP and the GP series shown before. One may note that the ACTR versions outperform the other algorithms and amongst them the performance of the version fed with the AP series input stands out better. Figure 1 shows the pictorial view of the same comparison of anytime performances across all the timepoints in the given time window.

Till now, we have shown the comparison of ACTR with the existing algorithms where it is clearly outperforming the others. Now, we compare the different runs of ACTR algorithm fed with different Contract series inputs amongst themselves to analyze their behavior. Table 6 shows the comparison of different runs of ACTR fed with different inputs, the AP series, the GP series, the Random series, and the default series (OACTR), when sampled at the intervals of the AP series. As one

Table 4 Comparison of OACTR with the existing algorithms on the 50 (sliding-tile) 24-puzzle benchmarks at uniformly distributed time intervals

	Top count versus **time** (min)							
Algorithm	**4**	**12**	**20**	**28**	**36**	**44**	**52**	**60**
OACTR	**46**	**47**	**46**	**44**	**43**	**44**	**43**	**44**
AWA*	6	5	7	9	8	8	7	6
BS_{500}	2	1	1	1	1	1	1	1
IW	1	2	1	1	0	1	1	1
ANA*	0	0	1	1	1	1	1	1
DFBB	0	0	0	0	0	0	0	0

Table 5 Comparison of the average anytime performances on the 50 puzzle benchmarks at uniformly distributed time intervals

	$\%\,Optimal\,Closeness$ versus **time** (min)							
Algorithm	**4**	**12**	**20**	**28**	**36**	**44**	**52**	**60**
ACTR_{AP}	87.5	89.7	**90.7**	**91.4**	**91.6**	**91.7**	**91.8**	**91.8**
ACTR_{GP}	**88.0**	**90.1**	**90.7**	91.1	91.2	91.4	91.5	**91.8**
OACTR	85.5	88.3	89.3	90.0	90.3	90.5	90.6	90.8
AWA*	77.7	80.2	81.6	82.4	82.6	82.8	83.0	83.3
BS_{500}	72.6	74.2	75.6	76.1	76.6	76.9	77.2	77.5
IW	73.4	77.0	77.6	78.0	78.2	79.3	79.5	79.9
ANA*	62.4	67.2	68.6	71.0	71.8	72.1	72.4	72.7
DFBB	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

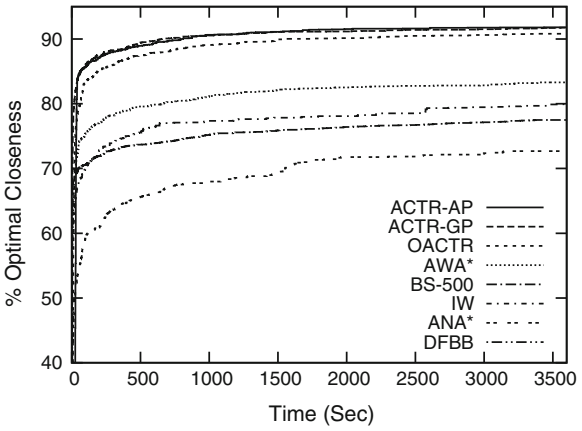

Fig. 1 Comparison of the average anytime performances on the 50 puzzle benchmarks

Table 6 Comparison of different runs of ACTR on the 50 (sliding-tile) 24-puzzle benchmarks at contract series of the AP series

	Top count versus **time** (min)							
Algorithm	**4**	**12**	**20**	**28**	**36**	**44**	**52**	**60**
ACTR_{AP}	26	33	**34**	**36**	**37**	**37**	**35**	32
ACTR_{GP}	**36**	**34**	32	33	34	34	34	**35**
ACTR_{RD}	26	27	25	27	31	32	31	32
OACTR	13	15	19	23	26	26	26	26

Table 7 Comparison of different runs of ACTR on the 50 (sliding-tile) 24-puzzle benchmarks at contract series of the GP series

	Top count versus **time** (min)						
Algorithm	**1**	**2**	**4**	**8**	**16**	**32**	**64**
ACTR_{AP}	31	27	26	31	**38**	**37**	33
ACTR_{GP}	**34**	**34**	**36**	**35**	35	33	**35**
ACTR_{RD}	32	24	26	34	25	28	33
OACTR	12	14	13	16	20	25	26

would expect/want, the ACTR run fed with the AP series input does well in most cases. Similarly, in Table 7, we observe that the ACTR run with the GP series input does well compared to others as the output sampling is measured against the GP series timepoints.

However, in Table 8, we observe that the ACTR runs with the AP series and the GP series inputs continue to dominate while we expect the run with the random series input to do well as the output sampling is measured against the random series timepoints. More viewpoints on such behavior are presented in Sect. 4.

Table 8 Comparison of different runs of ACTR on the 50 (sliding-tile) 24-puzzle benchmarks at contract series of the random series

	Top count versus **time** (min)							
Algorithm	**4**	**10**	**23**	**26**	**31**	**49**	**54**	**60**
$ACTR_{AP}$	26	31	**36**	**35**	**37**	**36**	**34**	32
$ACTR_{GP}$	**36**	**36**	32	33	33	33	**34**	**35**
$ACTR_{RD}$	26	31	26	25	27	32	32	32
OACTR	13	17	21	24	24	27	27	26

3.2 Travelling Salesman Problem

The first 50 symmetric TSPs (when sorted in increasing order of their sizes) from the traveling salesman problem library (TSPLIB) [13] are chosen 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 modification makes the search space of TSP a graph 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 state space).

Note that, a duplicate node can only exist in the same level as that of the new node under consideration (as the number of cities covered must be equal). Also, all the goal nodes are at a fixed depth, which helps reducing the number of goal node checks. Goal nodes at a known maximum depth m also means that when a expansion limit is reached at a particular level $l(< m)$ as per ACTR, none of the nodes belonging to the levels $< l$ need to be expanded in that iteration.

We repeat the exercise done in the case of Sliding-tile Puzzle Problem experiments in here as well. Firstly, we compare the performance of the proposed algorithm with that of the other algorithms when fed with different distributions of the timepoints across the time-window. Table 9 shows the comparison when ACTR is fed with the AP series input. BS_{100} indicates the results of Beam-stack search algorithm when run with beam-width $= 100$. The beam-width value is chosen after experimenting with several values such that neither the initial solution of the algorithm is delayed (due to large beam-width) nor the anytime performance is hampered (due to small beam-width). It can be noted that the proposed algorithm once again outperforms the other algorithms.

Table 9 Comparison of ACTR with the existing algorithms on the 50 TSP benchmarks when the contract series input is an AP series

	Top count versus **time** (min)							
Algorithm	**4**	**12**	**20**	**28**	**36**	**44**	**52**	**60**
ACTR	**36**	**40**	**43**	**41**	**42**	**41**	**41**	**41**
AWA*	28	31	27	27	27	28	28	28
BS_{100}	23	24	24	24	25	25	25	25
IW	21	18	19	19	19	20	19	19
ANA*	14	15	15	17	17	17	17	17
DFBB	10	10	11	11	11	11	11	11

Table 10 Comparison of ACTR with the existing algorithms on the 50 TSP benchmarks when the contract series input sequence is a GP series

	Top count versus **time** (min)						
Algorithm	**1**	**2**	**4**	**8**	**16**	**32**	**64**
ACTR	**40**	**40**	**44**	**44**	**43**	**43**	**43**
AWA*	25	27	28	27	25	26	26
BS_{100}	23	24	24	25	24	25	25
IW	19	19	18	17	19	18	18
ANA*	13	13	14	15	17	17	17
DFBB	10	10	10	11	11	11	11

Table 11 Comparison of ACTR with the existing algorithms on the 50 TSP benchmarks when the contract series input is generated randomly

	Top count versus **time** (min)							
Algorithm	**4**	**10**	**23**	**26**	**31**	**49**	**54**	**60**
ACTR	**37**	**41**	**43**	**41**	**42**	**41**	**41**	**41**
AWA*	28	31	27	27	27	27	28	28
BS_{100}	24	24	24	24	25	25	25	25
IW	20	17	19	19	19	19	19	19
ANA*	14	15	15	17	17	17	17	17
DFBB	10	10	11	11	11	11	11	11

Similarly, Tables 10 and 11 also indicate that the proposed algorithm comes on top when fed with the GP series input and the Random series input respectively.

Table 12 shows the comparison of the Oblivious Anytime Contract Search algorithm (OACTR) when sampled at uniformly distributed timepoints. The algorithm is run with a minimal contract of n in each iteration where n is the number of cities of the TSP instance. This strategy also results in best performance amongst the competing algorithms.

Now, we present the comparison of average anytime performances of the algorithms in terms of $\%\ Optimal\ Closeness$. Table 13 shows the comparison at uniformly distributed time intervals. ACTR_{AP} and ACTR_{GP} denote the runs of ACTR

Table 12 Comparison of OACTR with the existing algorithms on the 50 TSP benchmarks at the uniformly distributed time intervals

	Top count versus **time** (min)							
Algorithm	**4**	**12**	**20**	**28**	**36**	**44**	**52**	**60**
OACTR	**34**	**40**	**43**	**43**	**43**	**42**	**42**	**42**
AWA*	27	30	26	25	26	27	27	27
BS_{100}	25	24	25	24	25	25	25	25
IW	19	19	20	19	19	20	19	19
ANA*	14	15	15	17	17	17	17	17
DFBB	11	10	11	11	11	11	11	11

Table 13 Comparison of the average anytime performances on the 50 TSP benchmarks at uniformly distributed time intervals

	% Optimal Closeness versus **time** (min)							
Algorithm	**4**	**12**	**20**	**28**	**36**	**44**	**52**	**60**
$ACTR_{AP}$	97.8	98.2	98.4	98.4	98.4	98.4	98.5	98.5
$ACTR_{GP}$	**98.2**	**98.4**	**98.5**	**98.5**	**98.5**	**98.6**	**98.6**	**98.6**
OACTR	97.9	98.2	98.4	**98.5**	**98.5**	98.5	98.5	**98.6**
AWA*	97.5	97.8	97.8	97.8	97.8	98.0	98.0	98.0
BS_{100}	86.1	97.6	97.7	97.7	97.7	97.7	97.7	97.7
IW	97.2	97.8	97.8	98.0	98.0	98.0	98.1	98.1
ANA*	91.5	92.1	92.7	92.8	92.8	92.8	92.8	92.8
DFBB	90.0	90.3	90.5	90.6	90.7	90.7	90.7	90.8

algorithm with the input Contract series being the AP and the GP series shown before. One may note that the ACTR versions outperform the other algorithms and amongst them the performance of the version fed with the GP series input stands out better. Figure 2 shows the pictorial view of the same comparison of anytime performances

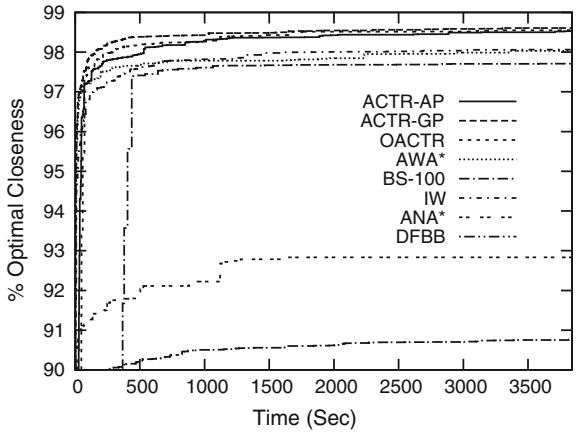

Fig. 2 Comparison of the average anytime performances on the 50 TSP benchmarks

across all the timepoints in the given time window. One may also note that the quality of the solutions reported is as high as 98.5 % on an average.

Finally, we present the comparison of different runs of ACTR when fed with different Contract series inputs. Table 14 shows the comparison when the outputs are sampled at the time intervals that match the given AP series input. We observe that all the runs are very competitive in this domain without a clear winner. An interesting thing to note here is that the oblivious version (OACTR) performs quite well in this domain.

Tables 15 and 16 show the comparison when the outputs of the algorithms are sampled at the intervals of the GP series and the random series, respectively. Once again, we note that there is no clear winner and the run with the GP series input seems to hold a bit of an advantage at some timepoints. This brings us to the discussion on the future scope for studying this problem and improving the proposed algorithms, which we discuss ahead.

4 Discussion and Conclusion

We build our discussion on the experimental observations of Tables 6, 7, and 8 and Tables 14, 15, and 16 where we observed that for some timepoints, the algorithm with the corresponding input series is not the best performer, which is against the expectations. There are several factors which contribute to such deviations which need to be studied and tuned in future. Broadly, the two major factors which impact

Table 14 Comparison of different runs of ACTR on the 50 TSP benchmarks at contract series of the GP series

	Top count versus **time** (min)							
Algorithm	**4**	**12**	**20**	**28**	**36**	**44**	**52**	**60**
ACTR_{AP}	35	39	**42**	39	42	42	**43**	40
ACTR_{GP}	38	**41**	41	**45**	**45**	42	42	43
ACTR_{RD}	**39**	38	**42**	41	42	**43**	42	41
OACTR	35	38	37	40	41	41	**43**	**44**

Table 15 Comparison of different runs of ACTR on the 50 TSP benchmarks at contract series of the GP series

	Top count versus **time** (min)						
Algorithm	**1**	**2**	**4**	**8**	**16**	**32**	**64**
ACTR_{AP}	32	32	36	38	38	38	35
ACTR_{GP}	**38**	33	**41**	**43**	40	**40**	38
ACTR_{RD}	32	**35**	36	41	40	39	36
OACTR	31	30	35	39	**41**	39	**40**

Table 16 Comparison of different runs of ACTR on the 50 TSP benchmarks at contract series of the random series

	Top count versus **time** (min)							
Algorithm	**4**	**10**	**23**	**26**	**31**	**49**	**54**	**60**
$ACTR_{AP}$	**45**	38	39	41	41	40	42	42
$ACTR_{GP}$	37	**40**	**45**	**46**	**46**	41	42	42
$ACTR_{RD}$	38	39	41	42	42	**4 2**	41	41
OACTR	35	**40**	39	39	41	**42**	**43**	**43**

the performance of the algorithm in this case are: (1) the maximum contract per iteration, and (2) the contract distribution scheme.

Regarding the maximum contract per iteration which is kept as a constant in our experiments (= 8 min), the same can be either increased gradually, in an AP series or a GP series, or can be learned dynamically as the algorithm progresses. This aspect needs to be studied further in future and how the same can be adapted to different domains.

The contract distribution scheme is a major aspect of the algorithm. In this paper, our scheme builds on the previous distributions and the number of nodes expanded so far. However, there is ample scope here to study entirely different and novel schemes and choose the best one. We have studied some other basic schemes such as, one which always distributes the contract equally amongst all levels, which did not perform well compared to the ones presented in the paper. The given scheme uses a pre-defined constant value for α which may also be learned/tuned during the run.

Coming up with a good dynamically learning framework for the contract distribution is an interesting problem. Also, one may explore other options inspired by the existing contract search algorithms that are parameter-free.

In conclusion, we proposed the problem of optimizing anytime performance of an algorithm according to a given input Contract series. An efficient algorithm is presented which distributes the contracts at various stages across different levels in terms of node expansion limits. Experimental results indicate that the proposed algorithm and its variations outperform some of the existing anytime algorithms consistently on the Sliding-tile Puzzle Problem and Traveling Salesperson Problem domains. This also highlights the strength underlying the proposed framework in terms of traditional anytime performance. Several interesting future research directions are also identified.

References

1. Aine, S., Chakrabarti, P.P., Kumar, R.: AWA* - A window constrained anytime heuristic search algorithm. In: M.M. Veloso (ed.) IJCAI, pp. 2250–2255 (2007).
2. Aine, S., Chakrabarti, P.P., Kumar, R.: Contract search: Heuristic search under node expansion constraints. In: ECAI, pp. 733–738 (2010).

3. Aine, S., Chakrabarti, P.P., Kumar, R.: Heuristic search under contract. Computational Intelligence **26**(4), 386–419 (2010).
4. Dean, T., Boddy, M.: An analysis of time-dependent planning. In: Proceedings of 6th National Conference on Artificial Intelligence (AAAI 88), pp. 49–54. AAAI Press, St. Paul, MN (1988).
5. Dionne, A.J., Thayer, J.T., Ruml, W.: Deadline-aware search using on-line measures of behavior. In: SOCS (2011).
6. 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).
7. Hiraishi, H., Ohwada, H., Mizoguchi, F.: Time-constrained heuristic search for practical route finding. In: H.Y. Lee, H. Motoda (eds.) PRICAI98: Topics in Artificial Intelligence, *Lecture Notes in Computer Science*, vol. 1531, pp. 389–398. Springer, Berlin Heidelberg (1998).
8. Korf, R.E., Felner, A.: Disjoint pattern database heuristics. Artif. Intell. **134**(1–2), 9–22 (2002).
9. Lawler, E.L., Wood, D.E.: Branch-and-bound methods: A survey. Operational Research **14**(4), 699–719 (1966).
10. Likhachev, M., Gordon, G.J., Thrun, S.: ARA*: Anytime A* with provable bounds on suboptimality. In: Advances in Neural Information Processing Systems 16. MIT Press, Cambridge, MA (2004).
11. Lowerre, B.: The Harpy Speech Recognition System. PhD thesis, Carnegie Mellon University (1976).
12. Norvig, P.: Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1992).
13. Reinelt, G.: TSPLIB - A traveling salesman problem library. ORSA Journal on Computing **3**, 376–384 (1991).
14. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach, 2 edn. Pearson, Education (2003).
15. Sturtevant, N.R., Felner, A., Likhachev, M., Ruml, W.: Heuristic search comes of age. In: AAAI (2012).
16. van den Berg, J., Shah, R., Huang, A., Goldberg, K.Y.: Anytime nonparametric A*. In: AAAI (2011).
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-05), pp. 90–98. Monterey, CA (2005).