The Thirteenth International Symposium
on Combinatorial Search (SoCS 2020)

A Guide to Budgeted Tree Search

Nathan R. Sturtevant
Department of Computing Science
University of Alberta, Canada
nathanst@ualberta.ca

Malte Helmert
Department of Mathematics and Computer Science
University of Basel, Switzerland
malte.helmert@unibas.ch

Abstract

Budgeted Tree Search (BTS), a variant of Iterative Budgeted Exponential Search, is a new algorithm that has the same performance as IDA* on problems where the search layers grow exponentially, but has far better performance than IDA* in other cases where IDA* fails. The goal of this paper is to provide a detailed guide to BTS with worked examples to make the algorithm more accessible to practitioners in heuristic search.

Introduction

Iterative Deepening A* (IDA*) (Korf 1985) is the classical algorithm for heuristic tree search as it is able to find optimal solutions while using memory linear in the search depth. Thus, IDA* has typically been used for searches in exponential domains, such as Rubik's Cube, that would not fit in memory. However, IDA* only runs in linear time relative to the size of the tree when the size of the iterative layers in the search grows exponentially. If the number of new nodes in each iteration only grows linearly, IDA* will incur a quadratic overhead, performing $\Theta(N^2)$ re-expansions of N nodes in the search tree.

Budgeted Tree Search (BTS), a depth-first version of Iterative Budgeted Exponential Search (IBEX), can be used as a replacement for IDA*. In the best case, where the search tree grows exponentially, BTS has no additional overhead over IDA*; they both perform $O(N)$ expansions. But, in the worst case problems for IDA*, BTS only requires $O(N \log(C^*/\epsilon))$ expansions, where C^* is the optimal solution cost and ϵ is the granularity of action costs (e.g., $\epsilon = 1$ for integer-cost problems and $\epsilon = 0.01$ for action costs with two decimal digits).

Our previously published work on BTS provides precise theoretical descriptions of the algorithm but lacks some details that would be helpful for anyone implementing the algorithm the first time. As a position paper, the purpose of this work is to provide a technical discussion of an implementation of the BTS algorithm, with detailed examples and pseudo-code. It is meant to be a companion to our original work (Helmert et al. 2019) and thus omits experimental results, a broad discussion of the literature and related algorithms, and detailed proofs of complexity and correctness. This paper focuses on what has not been published elsewhere: the full details needed for a correct implementation of BTS, along with a description of many of the low-level implementation details and examples of the algorithm running in practice.

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Background

We consider state-space search problems solved by heuristic tree search with an admissible heuristic (Pearl 1984; Russell and Norvig 2003). Our tree search algorithms assume a black-box interface to the state space supporting the following operations on states and actions:

- obtaining the *initial state* of the state space
- testing if a given state is a *goal state*
- determining the (finite) set of actions that are *applicable* in a given state
- *applying* an applicable action to modify a state
- determining the *cost* of an action (a non-negative number)

For additional efficiency, our pseudo-code also assumes that we can *undo* the application of an action to a state s to get back the state s. This allows our implementation to maintain a single copy of the state that is progressively modified during search. When such an operation is not available, successor states can be less efficiently generated as copies.

A *path* in the state space is defined by a sequence of actions that can be applied consecutively to the initial state. The *cost* of that path is the sum of costs of the actions in the path. In the tree search algorithms considered in this paper, there is a 1:1 correspondence between the search *nodes* generated by the algorithm and the paths it explores. A *solution* is a path that ends in a goal state. The objective of the tree search algorithm is to find an *optimal* solution, i.e., one with minimal cost. We do not discuss how algorithms determine unsolvability in this paper, but the considerations that apply are the same ones as for other tree search algorithms in the literature.

A heuristic tree search algorithm additionally requires an *admissible heuristic function* h that maps states s to non-negative numbers or infinity such that $h(s) \leq h^*(s)$, where

$h^*(s)$ is the optimal solution cost starting from state s (∞ if no solution from s exists). On a search node n that has reached state s, we define $f(n) = g(n) + h(s)$, where $g(n)$ is the path cost with node n (i.e., the cost occurred by the expansions that led to n).

A key ingredient of IDA* (and in modified form, also of IBEX) is the use of f-bounded depth-first searches. Given a numerical bound B, we conduct a depth-first search in the state space that *prunes* (ignores) all nodes n with $f(n) > B$. IDA* consists of a sequence of such f-bounded searches, where the initial bound is the heuristic value of the initial state, and every subsequent bound is the lowest f value of all nodes pruned in the previous search. The first solution encountered by IDA* is guaranteed to be optimal.

If h is not just admissible but also consistent (Dechter and Pearl 1985), the nodes expanded by IDA* (at least once) can be almost completely characterized by their f values. A node n will be expanded if $f(n) < C^*$, and it will not be expanded if $f(n) > C^*$. The behavior with $f(n) = C^*$ in general depends on tie-breaking in the last search layer of IDA*. In the following, we write N for the number of *relevant* nodes in the search tree, i.e., those with $f(n) \leq C^*$. If the search trees grow sufficiently rapidly between f-bounded searches, IDA* performs $O(N)$ expansions, but in the worst case it can perform $1+\cdots+N = N(N+1)/2 = \Theta(N^2)$ expansions. Note that our definition of relevant nodes includes those with $f(n) = C^*$, because in the worst case they will all be expanded.

The same analysis also applies to admissible but inconsistent heuristics, except that $f(n)$ must be replaced with the maximum f value of a node n and its ancestors in the search tree.

As mentioned previously, the BTS algorithm discussed in this paper is a specific implementation of the more general IBEX framework. The IBEX framework refers broadly to algorithms that use a specific combination of exponential and binary searches with an expansion budget to find optimal solutions. In addition to BTS, Budgeted Graph Search (BGS) is another instance of the IBEX framework that improves on the worst-case performance of A* (Helmert et al. 2019).

Iterative Deepening with an Oracle

To begin, we look at the expected behavior of IDA*, how it breaks down, and what information an oracle might provide to remedy the situation. In the following section we will show how the functionality of the oracle can be built in practice.

Consider the tree in Figure 1(a). This tree is built from the 3x2 5-tile sliding tile puzzle with the start state (5 4 3 2 1 0) and the goal state (0 1 2 3 4 5). Using a Manhattan Distance heuristic, IDA* can solve this problem with only 39 node expansions if all actions have the same cost of 1 – it performs iterations with f-limits of 11, 13 and 15, finding the solution cost 15 on the third iteration.

But, if the cost of moving a tile depends on the tile being moved the results are significantly different. Let the cost of an action moving tile t be $\frac{t+2}{t+1}$. So, moving the 1 tile costs 1.5, while moving the 5 tile costs 1.1667. (In our implementation all floating point comparisons are made with a tolerance of 1e-6.) With these costs, the full tree contains 100 relevant nodes (with f-cost no greater than the solution cost). If we solve the problem with IDA* and the (unit-cost) Manhattan distance heuristic, the total number of node expansions will be 3,793, which is close to the theoretical maximum of 100·101/2= 5,050 node expansions. IDA* requires 65 iterations to solve the problem, with an average of just 1.78 new expansions in each iteration.

Iteration	Nodes	f-cost
0	1	11.00
1	2	11.25
2	11	13.97
3	47	17.17
4	99	18.32
5	117	19.35

Table 1: The f-costs and node expansions when using an oracle for search.

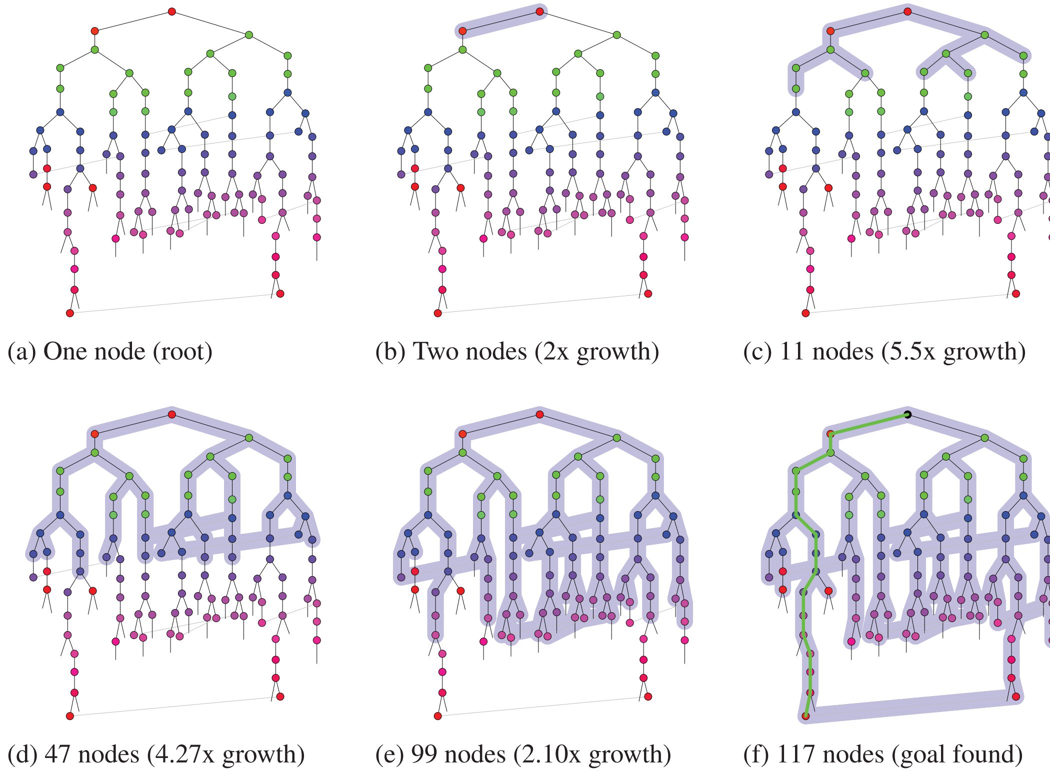
Consider, however, that we had an oracle that provided the next bound. That is, given any f-cost and the number of nodes, N_1, expanded for this f-cost, the oracle is able to provide a new f-cost for which an f-bounded DFS will perform at least $c_1 \cdot N_1$ and fewer than $c_2 \cdot N_1$ expansions, where $c_2 \geq c_1 > 1$ are parameters of the algorithm. That is, the oracle can provide a new f-cost that will ensure that the tree is growing exponentially from iteration to iteration. Using c_1 and c_2 makes it easier for the oracle to select the new f-cost, as there may be no f-cost that results in exactly constant growth of the tree.

The use of this oracle is shown in Figure 1 with bounds in Table 1. We use the parameters $c_1 = 2$ and $c_2 = 8$. In the trees in Figure 1 the states and branches within the f-bound have a thick line drawn behind them. The figure displays the state space, not the search tree, so there are transpositions in the tree indicated by light gray horizontal lines between different branches of the tree.

With the oracle, the first iteration uses an f-bound of 11 and performs a single node expansion. The second iteration uses a bound of 11.25 and performs 2 node expansions (in the allowed range of $1 \cdot [2, 8) = [2, 8)$). The third iteration jumps to a bound of 13.97 and performs 11 node expansions (again in the allowed range of $2 \cdot [2, 8) = [4, 16)$). This continues until the goal is found with cost of 19.35.

If such an oracle was available, each iteration would be guaranteed to grow by at least a factor of c_1 and no more than a factor of c_2. This would result in overall exponential growth and would guarantee that there would be no asymptotic overhead to the iterative search, as in IDA* in many unit-cost domains.

There are, however, two exceptional scenarios in which we must allow the oracle to deviate from the required growth rate in the range $[c_1, c_2)$. Firstly, if a solution is found in the next iteration, it is of course acceptable to grow the number of expansions by a smaller factor than c_1. This can be seen in the final iteration in Table 1, where the number of expan-



(a) One node (root) (b) Two nodes (2x growth) (c) 11 nodes (5.5x growth)

(d) 47 nodes (4.27x growth) (e) 99 nodes (2.10x growth) (f) 117 nodes (goal found)

Figure 1: Iterative deepening search with an oracle providing f-cost bounds.

sions only grows from 99 to 117. Secondly, it may be the case that there exists no f-bound with a growth rate in the desired range. In this case, and if the solution has not been found, the oracle is permitted to return the *smallest possible* f-bound that leads to a growth rate of at least c_1, even if this leads to a growth rate larger than c_2.

The key insight of BTS is that we can build an oracle that, given f_1 and N_1, can find the next bound f_2 with $O(c_2 \cdot N_1 \cdot \log((f_2 - f_1)/\epsilon))$ node expansions. Thus, by applying the oracle repeatedly we can increase the f-bound in a controlled manner until the optimal solution is found. The running time is dominated by the last iteration, where the goal is found, or by the penultimate iteration (the last unsuccessful one), resulting in an overall running time of $O(N \log(C^*/\epsilon))$ expansions.

In order to prepare the description of BTS, we first explain exponential search (Bentley and Yao 1976) in detail. We then show why exponential search cannot directly be used for the oracle, but how we can modify it to be suitable for the task. Finally, we show how we can take advantage of the specifics of heuristic and tree search to build an even more efficient version of BTS.

Exponential Search

Exponential search is an algorithm designed to find an entry with value e in a sorted array of unbounded length – or alternatively, if the element is not in the array, find the position at which the element would need to be inserted to maintain sorted order. There are no assumptions made about the distribution of values found in the array, only that they are sorted. Exponential search requires $O(\log i)$ accesses to the array if the entry is found (or could be inserted) at position i in the array. In particular, exponential search takes advantage of the sorted nature of the array. For each position that it queries in the array, it only tests whether the given entry is greater than or less than e. Exponential search checks exponentially growing indices until a value is found which is greater than or equal to e. It then uses a binary search, if needed, to find e once upper and lower bounds are explored in the exponential phase of the search. Although exponential search refers to the entire process, we often use the term "exponential search" to refer to the portion of the search where the indices are growing exponentially, which is distinct from the binary search once the upper and lower bounds are established.

We illustrate this in Figure 2. This example shows an array in which the search is looking for the marked element. The assumption is that each entry in the array contains a value, but we do not need to know the values – only the result of the comparison operators. Exponential search begins by testing the values at index 0, 1, 2, 4, etc., which all return a value of $<$. (Due to space limitations the arrows for 1 and 2 are not labeled.) When the 64th entry is reached and returns a value of $>$, it is then possible to do a binary search between 32 and 64, indicated by arrows below the array, until the comparison operator returns $=$ when the value at entry 44 matches the

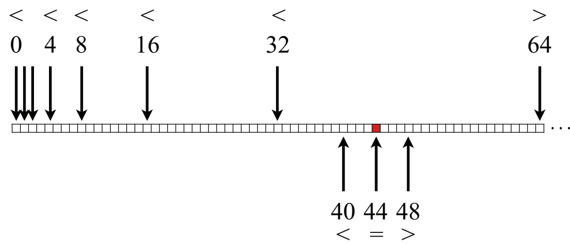


Figure 2: Exponential Search

value that was sought during the search.

Exponential search can be used on any data that is ordered over a set of indices. In particular, we can imagine that the indices of an array are all IEEE 754 32-bit floating point numbers in sorted order, and that the value of an index is the number of nodes that would be expanded by an f-bounded DFS using that f limit. (The number of expansions is not known *a priori* but can be computed by running an f-bounded DFS.) The key property is that an f-bounded DFS will never perform fewer node expansions with a larger f limit – the number of nodes expanded is monotonically non-decreasing with increasing f limits. Thus, node expansions are in sorted order relative to the indices of the array. Thus, exponential search can be used as an oracle for finding the next f-cost limit.

There is one limitation that prevents exponential search from being used efficiently. Exponential search will almost certainly sample indices in the array with $f > C^*$. An f-bounded DFS with $f > C^*$ may result in performing many more node expansions than are required to solve the problem, which would be inefficient. However, as we have noted, exponential search does not need to know how much greater a value is than the target value, only that it is greater. So, instead of running an f-bounded DFS, we can run an f-bounded *budgeted* DFS, where an upper bound (budget) is imposed on the number of node expansions. This search should terminate and return “$>$” once the node budget is exhausted. Using a budget ensures that the node expansions in any iteration will never exceed the number required to solve the problem by more than a constant factor.

Budgeted Tree Search

Exponential search has been adapted to the more generic IBEX algorithm, of which Budgeted Tree Search (BTS) is a specific example. BTS uses an exponential search on top of a depth-first search with f-limits and a node expansion budget. BTS is designed as a replacement for IDA* when the search layers are not guaranteed to grow exponentially. Specific design choices have been made to cause the algorithm to default to the same performance as IDA* if the tree does grow exponentially, and to only incur additional overhead if the growth of the search tree is not exponential. Note that our description of BTS here has minor differences from other descriptions (Helmert et al. 2019). These changes are primarily for pedagogical reasons in presenting examples and pseudo-code, but do not impact asymptotic performance.

Algorithm 1: BTS(c_1, c_2)

1 *solutionPath* $\leftarrow \varnothing$
2 *solutionCost* $\leftarrow \infty$
3 *budget* $\leftarrow 0$ // budget in node expansions
4 $i \leftarrow [h(\text{InitialState}()), \infty]$ // initial f interval
5 **while** $solutionCost > i.lower$ **do**
6   *solutionLowerBound* $\leftarrow i.lower$
7     $i.upper \leftarrow \infty$
8   **// 1. Regular IDA* Iteration**
9     $i \leftarrow i \cap \text{Search}(i.lower, \infty)$
10   **if** $nodes \geq c_1 \cdot budget$ **then**
11   *budget* $\leftarrow$ *nodes*
12   **continue**
13   **// 2. Exponential Search**
14     $\Delta \leftarrow 0$
15   **while** $((i.upper \neq i.lower) \wedge$
16     $(nodes < c_1 \cdot budget))$ **do**
17   *nextCost* $\leftarrow i.lower + 2^{\Delta}$
18         $\Delta \leftarrow \Delta + 1$
19   *solutionLowerBound* $\leftarrow i.lower$
20         $i \leftarrow i \cap \text{Search}(nextCost, c_2 \cdot budget)$
21   **// 3. Binary Search**
22   **while** $(i.upper \neq i.lower \wedge$
23     $\neg(c_1 \cdot budget \leq nodes < c_2 \cdot budget))$ **do**
24   *nextCost* $\leftarrow \frac{i.lower + i.upper}{2}$
25   *solutionLowerBound* $\leftarrow i.lower$
26         $i \leftarrow i \cap \text{Search}(nextCost, c_2 \cdot budget)$
27   *budget* $\leftarrow \max(nodes, c_1 \cdot budget)$
28   **if** $solutionCost = i.lower$ **then**
29   **return**

IDA* works by searching iterations where the f-limit of each successive iteration is determined by the minimum f-cost unexplored in the previous iteration. IDA* is made up of a top-level procedure that processes the results of each iteration and determines how a low-level DFS will search the next iteration. The low-level DFS is limited only by the maximum f-cost that can be expanded. The DFS also returns the minimum unexplored f-cost.

BTS initially does the same work of IDA*, but has three stages overall. The first stage is the IDA* search using an f-bounded DFS. If in this stage the tree fails to grow by a constant factor over the previous iteration, the second stage begins an exponential search. In this stage the f-bound grows exponentially until the cost of the DFS hits or exhausts the budget. If the budget is exhausted, then the third stage, a binary search, is used to find an f-limit that is within the budget.

Implementation

We provide detailed pseudo-code of BTS in Algorithms 1, 2 and 3. In this subsection we give an overview of the pseudo-code, describing many details of the algorithm.

BTS uses a budget for how many node expansions are

Algorithm 2: Search($costLimit, nodeLimit$)

```
1  f_below ← 0 // max f expanded below costLimit
2  f_above ← ∞ // next largest f
3  nodes ← 0 // nodes expanded
4  LimitedDFS(InitialState(), 0, costLimit, nodeLimit)
5  if nodes ≥ nodeLimit then
6  |  return [0, f_below]
7  else if f_below ≥ solutionCost then
8  |  return [solutionCost, solutionCost]
9  else
10 |  return [f_above, ∞]
```

Algorithm 3: LimitedDFS($currState, pathCost, costLimit, nodeLimit$)

```
1  currF ← pathCost + h(currState)
2  if solutionCost = solutionLowerBound then
3  |  return
4  else if currF > costLimit then
5  |  f_above ← min(f_above, currF)
6  |  return
7  else if currF ≥ solutionCost then
8  |  f_below ← solutionCost
9  |  return
10 else
11 |  f_below ← max(currF, f_below)
12 if nodes ≥ nodeLimit then
13 |  return
14 if IsGoalState(currState) then
15 |  solutionPath ← ExtractPath(currState)
16 |  solutionCost ← currF
17 |  return
18 acts ← GetActions(currState)
19 nodes ← nodes + 1
20 for a ∈ acts do
21 |  ApplyAction(currState, a)
22 |  LimitedDFS(currState, pathCost + Cost(a),
   |    costLimit, nodeLimit)
23 |  UndoAction(currState, a)
```

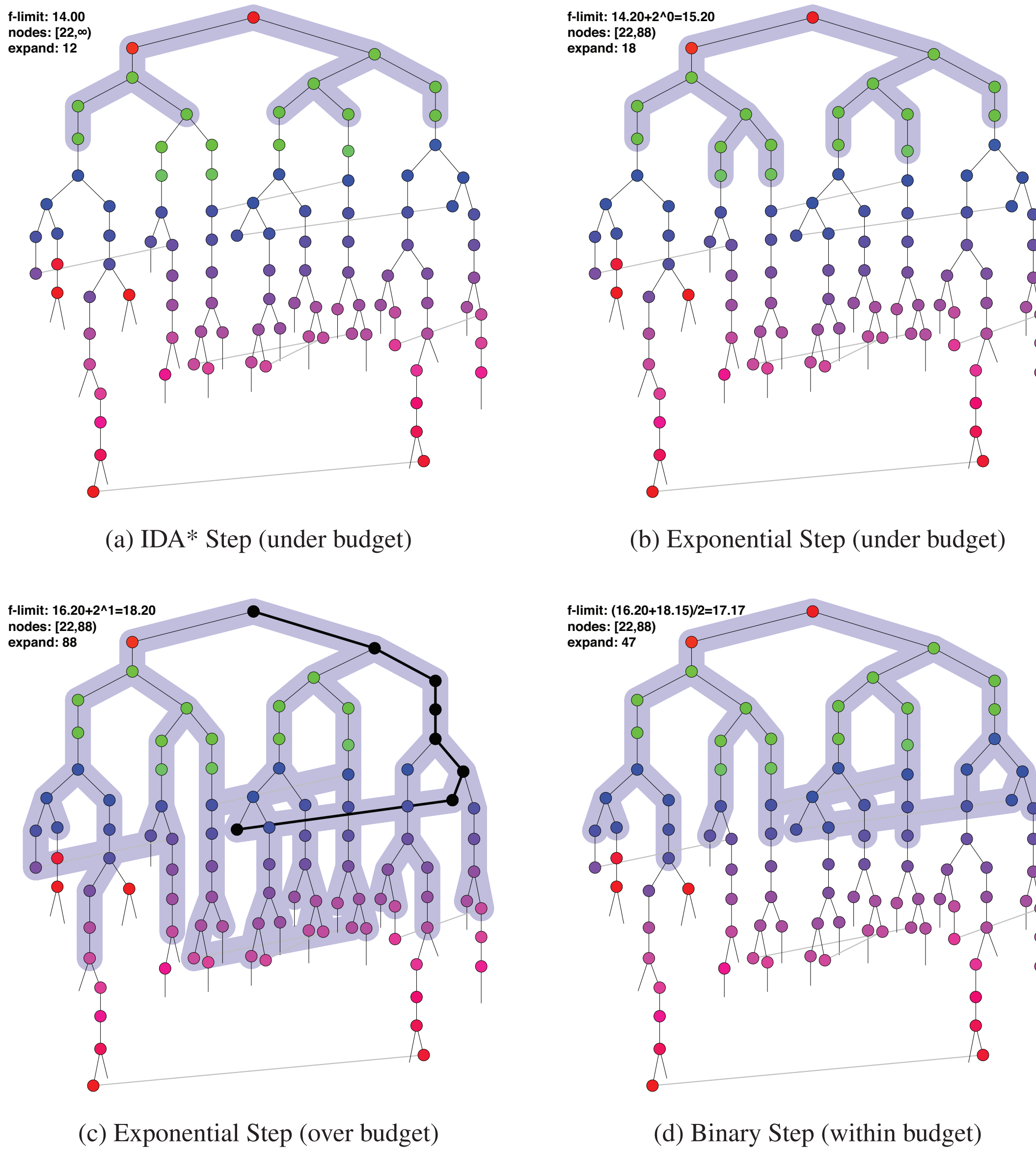
allowed in each iteration. This budget is initialized in Algorithm 1 line 3 and stores the total node expansions in the previous DFS. The budget for the next iteration is then between c_1 and c_2 times the previous budget. The goal is to find a target f-cost that results in expansions within this budget. The number of expansions in the last search is stored in a global variable named *nodes*. BTS also stores an interval (Algorithm 1 line 4) which contains the range of f-costs that may lead to exponential growth. At the beginning of each iteration the lower bound of the interval is the minimum f-cost not yet fully explored and the upper bound is infinity.

BTS uses a search procedure (Algorithm 2) which calls the low-level f-bounded budgeted DFS (Algorithm 3) and returns the range of possible values for the target f value. This search procedure is passed the f-bound and the budget, while the range of f-costs seen (f_{below} and f_{above}) are used as global variables. The interval that is returned indicates whether the budget was reached (meaning that the f-bound was too high), whether the solution was proven to be optimal, or whether the f-bound was too low and the minimum budget was not met.

The first stage of BTS (Algorithm 1 lines 8–12) just runs an f-limited search as in IDA*, calling the low-level search with an infinite budget. At this point the previous f-cost layer has been searched to completion, so C^* is not being overestimated and the next possible f-cost can be searched exhaustively. If the minimum budget is exceeded (line 10), the tree is growing exponentially, and thus we can move to the next iteration directly, as would IDA*.

If the initial search fails to hit the expansion budget, the exponential search (lines 13–20) begins to increase the f-cost bound in the low-level search exponentially (line 17) until the lower bound of the budget is exceeded. Unlike in the IDA* search, the exponential search uses a node budget to limit the maximum number of expansions by the low-level search. There are two points to note here. First, starting the exponential growth with $2^0 = 1$ assumes that the solution cost is at least 1; asymptotic guarantees would not hold for very small solution costs. (If this is a concern, one possible fix is to replace 2^Δ with $i.lower \cdot 2^\Delta$ in line 17. More generally, any multiplier between ϵ and C^* can be used instead of $i.lower$ to restore the guarantee.) Second, initializing Δ to 1 instead of 0 (line 14) may improve performance as it avoids the sequential steps seen in the beginning of the exponential search in Figure 2. But, more interestingly, it is possible to use more sophisticated methods (Burns and Ruml 2013) to try to guess the next f-value without breaking the theoretical guarantees.

When the exponential search completes, if the target f-value (within the budget) is found, the binary search will be skipped. Otherwise the binary search (lines 21–26) begins. While a binary search may seem straightforward, there are several notable cases to consider. In particular, assume that there is no target f-value within the budget. If the numerical resolution is significant, the binary search could iterate many steps on f-costs that do not correspond to actual f-costs in the tree. This is because the f-costs in the binary search are not derived from actual f-costs in the tree, as elsewhere in the search, but by taking the average of the upper and lower f-cost in the interval. To prevent this, the search returns both the smallest f-cost not expanded (f_{above}) and the largest f-cost expanded below the bound (f_{below}). When a search does not complete within the budget, f_{below} would have been sufficient to use up the budget, and thus this is returned instead of the f-cost limit that was used for the search (Algorithm 2 line 6). Thus, the binary search will not perform iterations at a higher resolution than the unique f-values in the tree. Because each step of the binary search decreases the size of the f-cost interval, eventually the interval will collapse to a single value – the minimum f-value that is above the budget. In this case BTS will begin its next iteration by searching that f-cost exhaustively.



(a) IDA* Step (under budget)

(b) Exponential Step (under budget)

(c) Exponential Step (over budget)

(d) Binary Step (within budget)

Figure 3: Iterative deepening search with an oracle providing f-cost bounds.

BTS terminates when a solution is both found and proven optimal. It does this by completing a search with the f-cost bound greater than or equal to the optimal solution cost. In this case it returns a collapsed interval at the solution cost (Algorithm 2 line 8), which will cause both the exponential and binary searches to terminate.

Because BTS skips some f-costs in the exponential and binary searches, it is possible that BTS will discover the goal with suboptimal solution cost or discover the goal before being able to prove that the solution cost is optimal. If BTS has found a solution but has not yet proven the optimality of the solution, BTS does not need to search states with f-costs greater than the current best solution (Algorithm 3 line 7).

This differs from IDA*, which is able to terminate as soon as it finds a solution. This is because it always searches with the lowest possible f-cost that has yet to be explored. To retain the behavior of IDA*, BTS must explicitly record the lowest possible f-cost so BTS can terminate if a solution is found with this cost. BTS records this prior to each search (Algorithm 1 lines 6, 19 and 25) and uses the value to terminate the low-level search in Algorithm 3 line 2 if a provably optimal solution is found.

Tree Example

We now turn to an example in Figure 3 to illustrate the behavior of BTS over a single iteration. This example shows how each of the steps of BTS work in practice. The problem is the same as before running on the 3x2 5-tile sliding tile

puzzle.

This example starts in Figure 3(a) where the previous iteration required 11 node expansions to complete with an f-limit of 13.97. The next f-bound (f_{above}) was 14.00, and so in the regular IDA* iteration an f-bounded DFS is run with infinite budget and a target of expanding at least $11 \cdot 2 = 22$ nodes. There are, however, only 12 nodes in the f-bounded sub-tree (the highlighted portion), so the minimum expansion target is not reached.

BTS then moves to the exponential search. The interval returned by the IDA* iteration was $[14.20, \infty]$ (f_{above} is 14.20), so the lower bound in the interval in Algorithm 1 is raised to 14.20. Because the expansion target was not reached, BTS now increases the cost bound exponentially. In this case, it will use $14.20 + 2^0 = 15.20$ as the next f-cost limit. The search tree with this limit is shown in Figure 3(b) and requires 18 node expansions, which is still under the node budget, which is targeting between 22 and 88 node expansions. Thus, BTS continues the exponential search.

The previous exponential search returned an interval of $[16.20, \infty]$, so the third iteration uses an f-bound of $16.20 + 2^1 = 18.20$. In Figure 3(c) the black line shows how far the search can proceed (from left to right) before it reaches the expansion budget. At this point the search is terminated, because the search budget has been reached. Although an f-bound of 18.20 was used, the maximum f-cost expanded was actually 18.15 (f_{below}), and thus the returned interval is $[0, 18.15]$. After taking the intersection with the previous interval, BTS now knows that it is looking for an f-cost in the interval $[16.20, 18.15]$. Note that because the search space shown in Figure 3(c) is a graph, not a tree, some states are expanded via multiple paths, including the state that was being expanded when the budget was exhausted.

Because the search budget was exceeded, BTS now moves to a binary search on f-costs. Given the search interval $[16.20, 18.15]$ an f-limit of $(16.20 + 18.15)/2 = 17.17$ is used for the next iteration shown in Figure 3(d). This iteration is completed using 47 node expansions, which is within the target budget, so 17.17 can be returned as the final f-cost for this iteration (see Table 1), and the budget will be updated to 47. f_{above} is 17.28, so the next iteration of BTS will start with an IDA* iteration of this cost.

In solving the entire problem, BTS requires just 564 node expansions, in contrast to IDA*, which requires 3,793.

Discussion

For completeness, it is worth pointing out that BTS can have worse performance than IDA* on some problem instances, particularly if an adversary is constructing the tree. For instance, suppose that BTS uses $c_1 = 2.0$ but in practice each layer grows by a factor of 1.99. In this case BTS will use the exponential and/or binary searches while IDA* will not. The overhead of these searches can result in BTS doing more work than IDA* in such cases. But, this overhead is still bounded by the worst-case of BTS, $O(N \log(C^*/\epsilon))$, and thus is far less than the overhead of IDA* when the search layers do not grow exponentially.

Conclusion

This paper has provided an in-depth look at Budgeted Tree Search, with detailed pseudo-code, examples, and algorithmic details that are useful for implementing BTS in practice. The intent of this paper is to make the algorithm easier to both teach and implement. The examples used in the paper are available from https://www.movingai.com/SAS/BTS/, including an interactive demo, movies, and image sequences. The pseudo-code used here is based on the implementations available from https://github.com/nathansttt/hog2/, with only small changes designed to increase the simplicity and readability.

References

Bentley, J. L., and Yao, A. C.-C. 1976. An almost optimal algorithm for unbounded search. *Information Processing Letters* 5(3):82–87.

Burns, E., and Ruml, W. 2013. Iterative-deepening search with on-line tree size prediction. *Annals of Mathematics and Artificial Intelligence* 69(2):183–205.

Dechter, R., and Pearl, J. 1985. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM* 32(3):505–536.

Helmert, M.; Lattimore, T.; Lelis, L. H. S.; Orseau, L.; and Sturtevant, N. R. 2019. Iterative budgeted exponential search. In Kraus, S., ed., *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*, 1249–1257. IJCAI.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Russell, S., and Norvig, P. 2003. *Artificial Intelligence — A Modern Approach.* Prentice Hall.