Iterative Budgeted Exponential Search

Malte Helmert[*1], **Tor Lattimore**[2], **Levi H. S. Lelis**[3], **Laurent Orseau**[2] and **Nathan R. Sturtevant**[4]
[1]Department of Mathematics and Computer Science, University of Basel, Switzerland
[2]DeepMind, London, UK
[3]Departamento de Informática, Universidade Federal de Viçosa, Brazil
[4]Department of Computing Science, University of Alberta, Edmonton, AB, Canada
malte.helmert@unibas.ch, {lattimore, lorseau}@google.com, levi.lelis@ufv.br, nathanst@ualberta.ca

arXiv:1907.13062v1 [cs.DS] 30 Jul 2019

Abstract

We tackle two long-standing problems related to re-expansions in heuristic search algorithms. For graph search, A* can require $\Omega(2^{n_*})$ expansions, where n_* is the number of states within the final f bound. Existing algorithms that address this problem like B and B' improve this bound to $\Omega(n_*^2)$. For tree search, IDA* can also require $\Omega(n_*^2)$ expansions. We describe a new algorithmic framework that iteratively controls an expansion budget and solution cost limit, giving rise to new graph and tree search algorithms for which the number of expansions is $O(n_* \log C^*)$, where C^* is the optimal solution cost. Our experiments show that the new algorithms are robust in scenarios where existing algorithms fail. In the case of tree search, our new algorithms have no overhead over IDA* in scenarios to which IDA* is well suited and can therefore be recommended as a general replacement for IDA*.

1 Introduction

There are two long-standing problems in heuristic search where existing algorithms struggle to balance the number of expansions and re-expansions performed in comparison to an oracle. One is in graph search, the other in tree search.

The first problem deals with admissible but inconsistent heuristics in *graph search*. With some caveats [Holte, 2010], A* with an admissible and consistent heuristic expands the minimum required number of states [Hart *et al.*, 1968; Dechter and Pearl, 1985]. However, with inconsistent heuristics it may expand exponentially more states than more cautious algorithms such as B [Martelli, 1977] and B' [Mérõ, 1984], which have a quadratic worst case.

The second problem is in heuristic *tree search* algorithms that use memory that grows only linearly with the search depth. In contrast, A* memory usage grows linearly with time and often exponentially with the depth of the search. To satisfy such low memory requirements, linear-memory tree search algorithms perform successive depth-first searches with an increasing limit on the cost. They also forgo global duplicate elimination, meaning that they do not detect if multiple paths from the initial state lead to the same state, which can lead to exponentially worse runtime compared to algorithms like A* when such duplicates are frequent. Hybrid algorithms that uses bounded memory for duplicate elimination are possible [Akagi *et al.*, 2010, for example].

IDA* [Korf, 1985] is a cautious linear-memory algorithm that increases the f-cost bound minimally (see also RBFS [Korf, 1993]). At each iteration, IDA* searches all nodes up to the f-cost bound. The minimum cost of the nodes pruned in one iteration becomes the cost bound for the next iteration. This approach ensures that the last cost bound will be exactly the minimum solution cost. This is efficient when the number of nodes matching the current cost bound grows exponentially with the number of iterations, as the total number of expansions will be dominated by the last iteration. In the worst case however, IDA* may expand only one new node in each iteration, leading to a quadratic number of (re-)expansions. Several methods have been developed to mitigate the re-expansion overhead of IDA* [Burns and Ruml, 2013; Sharon *et al.*, 2014; Hatem *et al.*, 2018; Sarkar *et al.*, 1991; Wah and Shang, 1994] by increasing the cost bound more aggressively at each iteration with the aim of achieving an exponential growth rate. However, with this approach the last cost bound can be larger than the minimum solution cost, which may incur an arbitrarily large performance penalty. For these algorithms, theoretical guarantees (when provided) require strong assumptions such as uniformity of the costs or branching factor [Hatem *et al.*, 2015, for example].

We propose a novel framework called Iterative Budgeted Exponential Search (IBEX) guaranteeing for both problems described above that the number of expansions is near-linear in the number of nodes whose cost is at most the minimum solution cost. This is achieved by combining two ideas: (1) a budget on the number of expansions and (2) an exponential search for the maximum f-cost that can be searched exhaustively within the given budget. This framework proves that no solution can be found for the current budget, and then doubles it until a solution is found. This ensures that the last budget is always within twice the minimum required budget, while amortizing the work on

[*]Alphabetical order. This paper is the result of merging two independent submissions to IJCAI 2019 [Orseau *et al.*, 2019; Sturtevant and Helmert, 2019].

early iterations due to the exponential growth of the budget.

We develop two simple and fast algorithms that enjoy near-linear expansion guarantees, propose a number of enhancements, show how the tree search and graph search problems can be reduced to our framework, and show that these algorithms perform at least as well as state-of-the-art algorithms on a number of traditional domains without exhibiting any of the catastrophic failure cases.

2 Heuristic Search Problems

A black box heuristic search problem is defined by a finite state space $\mathcal{S}$, a set of goal states $\mathcal{S}^* \subseteq \mathcal{S}$, a cost function $c : \mathcal{S} \times \mathcal{S} \to [0, \infty]$ and an initial state $s_{\text{init}} \in \mathcal{S}$. The successors of a state s are those states s' that can be reached by a finite-cost edge: $\texttt{succ}(s) = \{s' : c(s, s') < \infty\}$. This defines a directed graph $\mathcal{G} = (\mathcal{S}, E)$ where states correspond to vertices in the graph and the edges are the finite-cost successors $E = \{(s, s') : c(s, s') < \infty\}$. A path is a sequence of states $\pi = (s_t)_{t=1}^m$ with $s_1 = s_{\text{init}}$ and its cost is $g(\pi) = \sum_{t=1}^{m-1} c(s_t, s_{t+1})$, which may be infinite if there is no edge between adjacent states. The end state of path $\pi = (s_t)_{t=1}^m$ is $\texttt{state}(\pi) = s_m$ and its successor paths are $\texttt{succ}(\pi) = \{(s_t)_{t=1}^{m+1} : s_{t+1} \in \texttt{succ}(s_m)\}$. A state s is expanded when the function $\texttt{succ}(s)$ is called for s; a state s' is generated when $\texttt{succ}(s)$ is called creating $s' \in \texttt{succ}(s)$. A node corresponds to a single path π from the root of a search tree. Expanding a node corresponds to expanding $\texttt{state}(\pi)$ and generating all the corresponding successor paths (nodes). Search algorithms may expand the same state multiple times because multiple nodes might represent the same state. Let $\pi^*(s) = \text{argmin}_{\pi:\texttt{state}(\pi)=s}\, g(\pi)$ be a least-cost path to state s and $g^*(s) = g(\pi^*(s))$ be the cost of such a path. Let Π^* be the set of all paths from the initial state to all goal states. Then, the cost of a least-cost path to a goal state is $C^* = \min_{\pi \in \Pi^*} g(\pi)$. The objective is to find a least-cost path from the initial state to a goal state.

Let $h^*(s)$ be the minimal cost over all paths from s to any goal state. A heuristic is a function $h : \mathcal{S} \to [0, \infty]$ that provides an estimate of h^*. A heuristic is admissible if $h(s) \leq h^*(s)$ for all states $s \in \mathcal{S}$ and consistent if $h(s) \leq h(s') + c(s, s')$ for all pairs of states s, s'. The f-cost of a path is $f(\pi) = g(\pi) + h(\texttt{state}(\pi))$. Note that if $s = \texttt{state}(\pi)$ is a goal state and the heuristic is admissible we must have $h(s) = 0$.

We say that a search algorithm is a *graph search* if it eliminates duplicates of states generated by the algorithm; otherwise it is called a *tree search*.

2.1 Graph Search

With a consistent heuristic, f-costs along a path are non-decreasing, thus a graph search algorithm must expand all states in the graph with $f(s) = g^*(s) + h(s) < C^*$. In this setting, A* has an optimal behaviour [Dechter and Pearl, 1985]. When the heuristic is admissible but inconsistent, for comparing algorithms one could consider the ideal number of nodes that A* would expand if the heuristic was made consistent. Unfortunately, not only does there exist no optimal algorithm for this case [Mérõ, 1984], but it can even be shown that *all* algorithms may need to expand exponentially too many nodes in some cases (see Appendix E). Hence we focus our attention on the following relaxed notion of optimality. Let $n_{G*} = |\{s : \min_{\pi:\texttt{state}(\pi)=s} \max_{s' \in \pi(s)} f(s') \leq C^*\}|$ be the number of states that can be reached by a path along which all states have f-cost at most C^*; this is the definition used by Martelli [1977]. Then, there exist problems where A* performs up to $\Omega(2^{n_{G*}})$ expansions [Martelli, 1977]. This limitation has been partially addressed with the B [Martelli, 1977] and B' [Mérõ, 1984] algorithms for which the number of expansions is at most $O(n_{G*}^2)$. We improve on this result with a new algorithm for which the number of expansions is at most $O(n_{G*} \log(C^*))$.

2.2 Tree Search

Tree search algorithms work on the tree expansion of the state space, where every path from s_{init} corresponds to a tree node. Consequently, states reached on multiple paths will be expanded multiple times.

We say a path π is necessarily expanded if $\max_{s' \in \pi} f(s') < C^*$ and possibly expanded if $\max_{s' \in \pi} f(s') \leq C^*$. A tree search algorithm must always expand all necessarily expanded paths and will usually also expand some paths that are possibly but not necessarily expanded. To avoid the subtleties of tie-breaking, we discuss upper bounds in terms of possibly expanded paths, leaving a more detailed analysis for future work. We write n_{T*} for the number of possibly expanded paths in a tree search.

In the worst case, IDA* may perform $\Omega(n_{T*}^2)$ expansions. To mitigate this issue, algorithms such as IDA*_CR [Sarkar *et al.*, 1991] and EDA* [Sharon *et al.*, 2014] increase the f-cost bound more aggressively. These methods are effective when the growth of the tree is regular enough, but can fail catastrophically when the tree grows rapidly near the optimal f-cost, as will be observed in the experiments. We provide an algorithm that performs at most a logarithmic factor more expansions than n_{T*} and uses memory that is linear in the search depth.

Notation. The natural numbers are $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$ and $\mathbb{N}_1 = \{1, 2, 3, \ldots\}$. For real-valued x and a let $\lceil x \rceil_{\geq a} = \max\{a, \lceil x \rceil\}$ and similarly for $\lfloor x \rfloor_{\geq a}$.

3 Abstract View

We now introduce a useful abstraction that allows us to treat tree and graph search in a unified manner. Tree search is used as a motivating example. The problem with algorithms like EDA* that aggressively increase the f-cost limit is the possibility of a significant number of wasted expansions once the f-cost limit is above C^*. The core insight of our framework is that this can be mitigated by stopping the search if the number of expansions exceeds a budget and slowly increasing the budget in a careful manner.

A depth-first search with an f-cost limit and expansion budget reveals that either (a) the expansion budget was insufficient to search the whole tree with f-cost smaller or equal to the limit, or (b) the expansion budget was sufficient. In the latter case, if the goal is found, then the algorithm

can return a certifiably optimal solution. Furthermore, when the budget is insufficient the largest f-cost of a node visited by the search serves as an upper bound on the largest f-cost for which the budget will be exceeded. When the budget is sufficient, the smallest f-cost in the fringe is a lower bound on the same. This information means that combining exponential search [Bentley and Yao, 1976] with repeated depth-first searches with a varying f-cost limit and fixed expansion budget can be used to quickly find a solution if the budget is sufficient to expand all nodes with f-cost less than C^* and otherwise produce a certificate that the budget is insufficient, a process we explain in detail in Section 5.

Based on this idea, the basic version of our new algorithm operates in iterations. Within each iteration the algorithm makes multiple depth-first searches with a fixed expansion budget and varying f-cost limits. An iteration ends once the algorithm finds the optimal solution and expands all paths with f-cost less than C^*, or once it can prove that the present expansion budget is insufficient to find the optimal solution. At the end of the iteration the expansion budget is doubled.

In what follows we abstract the search procedure into a query function that accepts as input an f-cost limit and an expansion budget and returns an interval that contains the smallest f-cost limit for which the budget is insufficient or throws an exception with an optimal solution if the f-cost limit is at least C^* and the expansion budget is larger or equal to the number of nodes with f-cost less than the limit.

Formal model. We consider an increasing list A of real numbers $v \geq 1$, possibly with repetition. Define a function $n : [1, \infty) \to \mathbb{N}_0$ by $n(C) = |\{v \in A : v \leq C\}|$, where multiple occurrences are counted separately. Next, let $C^* \in A$ and $n_* = n(C^*)$. In our application to tree search, A is the list of all node f values, including duplicates, and $n(C)$ is the number of paths in the search tree for which the f values is at most C and C^* is the cost of the optimal solution. We can require that all f values are at least 1 with no loss of generality: if $h(s_{\text{init}}) < 1$ we introduce an artificial new initial state with heuristic value 1 and an edge of cost $1 - h(s_{\text{init}})$ from the new state to s_{init}. This shifts all path costs by at most 1, so if C' is the original optimal solution cost, we have $C^* \leq C' + 1$.

Query functions. Let $C_{\texttt{crit}}(b) = \min\{v \in A : n(v) > b\}$ be the smallest value in A for which expansion budget b is insufficient. We define three functions, $\texttt{query}_{\text{lim}}$, $\texttt{query}_{\text{int}}$ and $\texttt{query}_{\text{ext}}$, all accepting as input an f-cost limit C and expansion budget b. A call to any of the functions makes at most $\min\{b, n(C)\}$ expansions and throws an exception with an optimal solution if $n_* \leq n(C) \leq b$. Otherwise all three functions return an interval containing $C_{\texttt{crit}}(b)$ on which we make different assumptions as described next. The abstract objective is to make a query that finds an optimal solution using as few expansions as possible.

Limited feedback. In the limited feedback model the query function returns an interval that only provides information about whether or not the expansion budget b was smaller or larger than $n(C)$:

$$\texttt{query}_{\text{lim}}(C, b) = \begin{cases} [C, \infty] & \text{if } C < C_{\texttt{crit}}(b) \quad \text{budget sufficient}, \\ [1, C] & \text{if } C \geq C_{\texttt{crit}}(b) \quad \text{budget exceeded}. \end{cases}$$

Integer feedback. In many practical problems, the list A only contains integers. In this case we consider the feedback model:

$$\texttt{query}_{\text{int}}(C, b) = \begin{cases} [\lfloor C \rfloor + 1, \infty] & \text{if } C < C_{\texttt{crit}}(b), \\ [1, C] & \text{if } C \geq C_{\texttt{crit}}(b). \end{cases}$$

The discrete nature of the returned interval means that if $C_{\texttt{crit}}(b) \in [C, C+1]$, then

$$\texttt{query}_{\text{int}}(C, b) \cap \texttt{query}_{\text{int}}(C+1, b) = \{C_{\texttt{crit}}(b)\},$$

In other words, the exact value of $C_{\texttt{crit}}(b)$ can be identified by making queries on either side of an interval of unit width containing it. By contrast, $\texttt{query}_{\text{lim}}$ cannot be used to identify $C_{\texttt{crit}}(b)$ exactly.

Extended feedback. For heuristic search problems the interval returned by the query function can be refined more precisely by using the smallest observed f-cost in the fringe and largest f-cost of an expanded path. Define $A_>(C) = \min\{v \in A : v > C\}$ and $A_\leq(C) = \max\{v \in A : v \leq C\}$. When $C \in A$ we define $\delta(C) = A_>(C) - A_\leq(C)$ and

$$\delta_{\min} = \min\{\delta(C) : C \leq C^*, C \in A\}.$$

These concepts are illustrated in Fig. 1. In the extended feedback model, when the expansion budget is sufficient the response of the query is the interval $[A_>(C), \infty]$. Otherwise the query returns an interval $[1, v]$ where v is any value in $A \cap [C_{\texttt{crit}}(b), C]$ (for example $v = A_\leq(C)$):

$$\texttt{query}_{\text{ext}}(C, b) = \begin{cases} [A_>(C), \infty] & \text{if } C < C_{\texttt{crit}}(b), \\ [1, v], \text{with } v \in [C_{\texttt{crit}}(b), C] \cap A & \text{if } C \geq C_{\texttt{crit}}(b). \end{cases}$$

In tree search the value of v when $C \geq C_{\texttt{crit}}(b)$ is the largest f-cost over paths expanded by the search, which may depend on the expansion order. As for integer feedback, the information provided by extended feedback allows the algorithm to prove that an expansion budget is insufficient to find a solution.

Summary of results. In the following sections we describe algorithms for all query models for which the number of node expansions is at most a logarithmic factor more than n_*. The limited feedback model is the most challenging and is detailed last, while the extended feedback model provides the cleanest illustration of our ideas. The logarithmic factor depends on C^* and δ_{min} or $\delta(C^*)$. The theorems are summarized in Table 1, with precise statements given in the relevant sections.

Overview. In the next section we implement $\texttt{query}_{\text{ext}}$ for tree search and for graph search (Section 4). We then introduce a variant of exponential search that uses the query function to find the 'critical' cost for a given budget (Section 5). Our main algorithm (IBEX) uses the exponential search with a growing budget an optimal solution is found (Section 6). The DovIBEX algorithm is then provided to deal with the more general limited feedback setting (Sections 7 and 8).

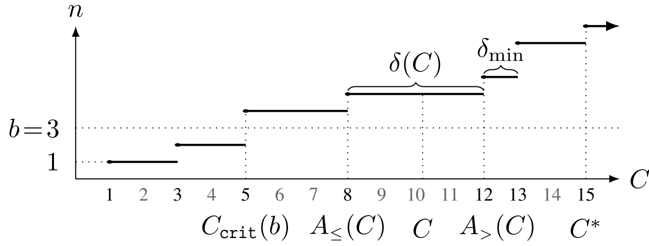

Figure 1: The function $n(\cdot)$, generated by $b = 3$, $C = 10.2$, $C^* = 15$ and $A = [1, 3, 5, 5, 8, 12, 13, 13, 15]$. $C_{\texttt{crit}}(b) = 5$ is the smallest value in A for which there are more than $b = 3$ values of at most 5. The largest value at most C in A is $A_{\leq}(C) = 8$, and $A_{>}(C) = 12$ is the next value in A. We also have $\delta_{\min} = 1$ and $\delta(C) = 4$. Example queries include: $\texttt{query}_{\text{lim}}(C = 4, b = 3) = [4, \infty]$ and $\texttt{query}_{\text{ext}}(C = 4, b = 3) = [5, \infty]$ and $\texttt{query}_{\text{ext}}(C = 9, b = 3) = [1, 8]$.

Limited feedback	$O(Z \log Z), Z = n_* \log\left(\frac{C^*}{\delta(C^*)}\right)$
Extended feedback	$O\left(n_* \log\left(\frac{C^*}{\delta_{\min}}\right)\right)$
Integer feedback	$O\left(n_* \log(C^*)\right)$

Table 1: Number of expansions in the worst case of our algorithms for the different types of feedback.

4 Reductions

We now explain how to reduce tree search and graph search to the abstract framework and implement $\texttt{query}_{\text{ext}}$ for these domains. These query functions will be used in the next sections as part of the main algorithms. Recall that the number of expansions performed by $\texttt{query}(C, \infty)$ must be at most $n(C)$ and $n(\cdot)$ is non-decreasing and that $n_* = n(C^*)$.

The list A is composed of the f-costs of the nodes encountered during the search. Recall from our problem definition that each node corresponds to a path π, and that expanding a node corresponds to expanding a state $s = \texttt{state}(\pi)$. For a search cost-bounded by C, let the fringe be all generated nodes with $f(\pi) > C$. Also, let the set of visited nodes be the generated nodes such that $f(\pi) \leq C$.

Tree search. For tree search we implement $\texttt{query}_{\text{ext}}$ in Algorithm 1, which is a variant of depth-first search with an f-cost limit C and expansion budget b. The algorithm terminates before exceeding the expansion budget and tracks the smallest f-cost observed in the fringe and the largest f-cost of any visited node, which are used to implement the extended feedback model. Thus, for an f-cost bound C, $n(C) = |\{\pi : \max_{s' \in \pi} f(s') \leq C\}|$. When an optimal solution is found, the algorithm throws an exception, which is expected to be caught and handled by the user. The function $\texttt{query}_{\text{lim}}$ could be implement like $\texttt{query}_{\text{ext}}$ without needing to track min_fringe and max_expanded, but that would throw away valuable information.

Graph search. In graph search, $\texttt{query}_{\text{ext}}$ is implemented in a similar way as Algorithm 1, but DFS is replaced with graph_search, which appears in Algorithm 7 (Appendix A). The graph_search function is equivalent to BFIDA* [Zhou and Hansen, 2004] with the breadth-first search replaced with Uniform-Cost Search (UCS) [Russell and Norvig, 2009; Felner, 2011], using an f-cost limit C on the generated nodes and tracking the maximum f-cost among visited states and the minimum f-cost in the fringe. As in UCS, states are processed in increasing g-cost order. Since g is non-decreasing, states are not expanded more than once in each call to Algorithm 7. Therefore the number of expansions is at most $n(C) = |\{s : \min_{\pi:\texttt{state}(\pi)=s} \max_{s' \in \pi} f(s') \leq C\}|$ and the number of expansions made by $\texttt{query}(C^*, \infty)$ is at most $n(C^*) = n_{G*}$ as required.

Algorithm 1 Query with extended feedback for tree search

```
1  def query_ext(C, budget):
2    data.min_fringe = ∞ # will be > C
3    data.max_visited = 0 # will be ≤ C
4    data.expanded = 0 # number of expansions
5    data.best_path = none # f(none) = ∞
6    try:
7      DFS(C, budget, {s_init}, data)
8    catch "budget exceeded":
9      return [1, data.max_visited]
10   if data.best_path ≠ none: # solution found
11     throw data # to be dealt with by the user
12   return [data.min_fringe, ∞]
13 # data: return info, passed by reference
14 # π: path
15 def DFS(C, budget, π, data):
16   if f(π) > C:
17     data.min_fringe = min(data.min_fringe,
         f(π))
18     return
19   data.max_visited = max(data.max_visited,
        f(π))
20   if f(π) ≥ f(data.best_path):
21     return # branch and bound
22   if is_goal(state(π)):
23     data.best_path = π
24     # Here we could throw the solution if its
25     # cost is equal to a known lower bound
26     return
27   if data.expanded == budget:
28     throw "budget exceeded"
29   data.expanded++
30   for s' ∈ succ(state(π)):
31     DFS(C, budget, π + {s'}, data)
```

5 Exponential Search

With the reductions out of the way, we now introduce a budgeted variant of exponential search [Bentley and Yao, 1976], which is closely related to the bracketed bisection method [Press *et al.*, 1992, §9].

Algorithm 2 accepts as input a budget b, an initial cost limit $\texttt{start} \leq C_{\texttt{crit}}(b)$ and a function $\texttt{query} \in$

Algorithm 2 Exponential search with budgeted queries

```
1  def exp_search(start, b, query):
2    low = start
3    high = ∞
4    loop:
5      if high == ∞:
6        C = 2×low # exponential phase
7      else:
8        C = (low + high) / 2 # binary phase
9      [low, high] = [low, high] ∩query(C, b)
10   until low == high
11   return low
```

$\{\mathtt{query}_{\text{ext}}, \mathtt{query}_{\text{int}}, \mathtt{query}_{\text{lim}}\}$. The algorithm starts by setting `low` = `start` and initiates an exponential phase where `low` is repeatedly doubled until `query`$(2 \times$ `low`$, b)$ has insufficient budget. The algorithm then sets `high` $= 2 \times$ `low` and performs a binary search on the interval [`low`, `high`] until `low` = `high`. See Fig. 2 for an illustration.

The discrete structure in the integer and extended feedback models ensures that the algorithm halts after at most logarithmically many queries and returns $C_{\mathtt{crit}}(b)$. In the limited feedback model the algorithm generally does not halt, but will make a terminating query if $b \geq n_*$. These properties are summarized in the next two propositions, which use the following definition:

$$n_{\text{exp}}(\varepsilon, x, \Delta) = 1 + \left\lceil \log_2 \left(\frac{x}{\varepsilon}\right) \right\rceil_{\geq 1} + \left\lfloor \log_2 \left(\frac{x}{\Delta}\right) \right\rfloor_{\geq 0} .$$

This is an upper bound on the number of calls to query needed when starting at `start` $= \varepsilon$, finding a upper bound `high` $\geq x$ and then reducing the interval [`low`, `high`] to a size at most Δ (leading to a query within that interval). Recall that making a query with cost limit C and expansion budget b will find an optimal solution if $n_* \leq n(C) \leq b$.

Proposition 1. Suppose $b \geq n_*$. Then for any feedback model Algorithm 2 makes a query that terminates the interaction after at most $n_{\text{exp}}(\mathtt{start},\ C^*,\ C_{\mathtt{crit}}(b) - C^*)$ calls to query.

Proposition 2. Suppose $b < n_*$. Then, for the extended feedback model, Algorithm 2 returns $C_{\mathtt{crit}}(b)$ with at most $n_{\text{exp}}(\mathtt{start},\ C_{\mathtt{crit}}(b),\ \delta_{\text{min}})$ queries.

6 Iterative Budgeted Exponential Search

The Iterative Budgeted Exponential Search (IBEX) algorithm uses the extended query model, which is available in our applications to tree and graph search. Algorithm 3 initializes a lower bound on the optimal cost with the lowest value in the array $C_1 = \min A$; we now denote this quantity by $C_{\min}$. It subsequently operates in iterations $k \in \mathbb{N}_0$. In iteration k it sets the budget to $b_k = 2^k$ and calls $C_{k+1} = \mathtt{exp_search}(C_k, b_k, \mathtt{query}_{\text{ext}})$ to obtain a better lower bound.

Theorem 3. The number of expansions made by Algorithm 3 is at most

$$4 n_* n_{\text{exp}}(C_{\min}, C^*, \delta_{\text{min}}) = O\left(n_* \log\left(\frac{C^*}{\delta_{\text{min}}}\right)\right) .$$

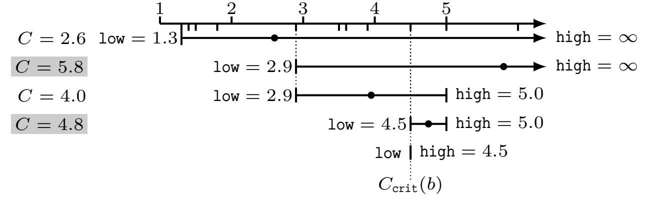

Figure 2: The four queries made by `exp_search` with the extended feedback model and `start` $= 1.3$ and budget `b` $= 7$. The ticks below the x-axis indicate the elements of $A = [1.4, 1.5, 1.8, 2.3, 2.9, 3.5, 3.6, 3.9, 4.5, 5, 6]$. The small circles are the values of C in each query. In the first call to $\mathtt{query}_{\text{ext}}$ the budget was sufficient and so `low` is set to $A_>(C) = 2.9$, which is doubled to produce the next query. In the second call to query, $C = 5.8$ leads to an insufficient budget and then `high` is set to 5.0. In the third query $C_{\mathtt{low}}$ is increased to 4.5, which is $C_{\mathtt{crit}}(b)$. In the fourth query the budget is insufficient and the algorithm halts.

Proof. Define $r_1 = n_{\text{exp}}(C_{\min}, C^*, \delta_{\text{min}})$. Let $k_* = \lceil \log_2 n_* \rceil$ be the first iteration k for which $b_k \geq n_*$. Proposition 2 shows that for iterations $k < k_*$ the number of queries performed in the call to `exp_search` is at most $n_{\text{exp}}(C_k, C_{\mathtt{crit}}(b_k), \delta_{\text{min}}) \leq r_1$. Proposition 1 shows that the game ends during iteration k_* after a number of queries bounded by $n_{\text{exp}}(C_{k_*}, C^*, C_{\mathtt{crit}}(b_{k_*}) - C^*) \leq r_1$. Since each call to query with budget b_k expands at most $b_k = 2^k$ nodes, the total number of expansions is bounded by $\sum_{k=1}^{k_*} 2^k r_1 \leq 2^{k_*+1} r_1 \leq 2^{2+\log_2 n_*} r_1 = 4 n_* r_1$. □

Remark 4. Algorithm 3 also works when $\mathtt{query}_{\text{ext}}$ is replaced by $\mathtt{query}_{\text{int}}$, and now $\delta_{\text{min}} = 1$.

When used for graph search, we call the resulting IBEX variant Budgeted Graph Search (BGS), and for tree search Budgeted Tree Search (BTS).

Remark 5. Observe that if DFS or graph_search are called with `budget` $\geq n(C) \geq n_*$, it throws an optimal solution. Since the state space is finite, both BTS and BGS return an optimal solution if one exists. If no solution exists, BGS will exhaust the graph and return "no solution", but BTS may run forever unless additional duplicate detection is performed.

7 Uniform Budgeted Scheduler

While IBEX handles the integer and extended feedback models, it cannot handle the limited feedback model. This is handled by a new algorithm, presented in the next section, using a finely balanced dovetailing idea that we call the Uniform Budgeted Scheduler (UBS, see Algorithm 4) and

Algorithm 3 Iterative Budgeted Exponential Search

```
1 def IBEX():   # simple version
2   C_1 = C_min
3   for k = 1,2,...
4     b_k = 2^k
5     C_{k+1} = exp_search(C_k, b_k, query_ext)
```

Algorithm 4 Uniform Budgeted Scheduler

```
1 def UBS(T, run_prog):
2   q = make_priority_queue(T)
3   q.insert((1, 1))  # k=1, r=1
4   while not q.empty():
5     # Remove prog of minimum T cost
6     (k,r) = q.extract_min()
7     budget = T(k,r) − T(k,r − 1)
8     if run_prog(k, budget) != "halted":
9       q.insert((k,r + 1))
10    if r == 1:
11      q.insert((k + 1,1))
12  return none
```

takes inspiration from Luby *et al.* (1993) speedup algorithm. UBS runs a growing and unbounded number of programs in a dovetailing fashion, for varying segments of *steps*. The notion of *step* is to be defined by the user; in heuristic search we take it to be a single node expansion. During one segment, the selected program can make arbitrary computations but must use no more steps than its current budget. Program k halts when it reaches exactly τ_k steps, which may be infinite. UBS maintains a priority queue of pairs (program index k, segment number r), initialized with $(1,1)$ and ordered by a function $T : \mathbb{N}_1 \times \mathbb{N}_0 \to \mathbb{N}_0$ with $T(k,r) < T(k,r+1)$ and $T(k,r) \leq T(k+1,r)$ for all (k,r) and $T(k,0) = 0$ for all k. In each iteration UBS removes the T-minimal element (k,r) from the front of the queue and calls run_prog(k,b) with $b = T(k,r) - T(k,r-1)$, which means that program k is being run for its rth segment with a budget of b steps, leading for program k to a total of at most $T(k,r)$ steps over the r segments. If this does not cause program k to halt, then $(k,r{+}1)$ is added to the priority queue. Finally, if $r = 1$, then $(k+1,1)$ is added too. The function run_prog is defined by the user and may store and restore the state of program k as well as allow access to a shared memory space.

The monotonicity assumptions on T mean that UBS is essentially executing program/segment pairs (k,r) according to a Uniform Cost Search where a pair (k,r) is a node in the search tree with cost $T(k,r)$ (Fig. 3). In this sense UBS tries (asymptotically) to maintain a uniform amount of steps used among all non-halting programs.

Let $T_{\text{UBS}}(k,r)$ be the number of steps used by UBS after executing (k,r). $T_j(k,r) = \max\{T(j,m) : T(j,m) \leq T(k,r), m \in \mathbb{N}_0\}$ is an upper bound on the number of steps used by program j when UBS executes (k,r).

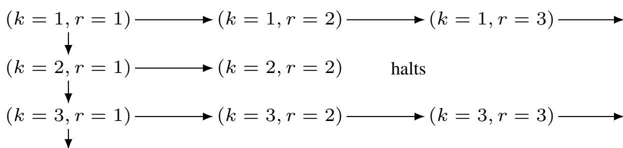

Figure 3: An example tree used by UBS. Program $k = 2$ halts after $\tau_2 \leq T(2,2)$ steps.

Theorem 6. For any (k,r) with $T(k,r-1) < \tau_k$,

$$T_{\text{UBS}}(k,r) \leq \sum_{j \in \mathbb{N}_1} \min\{\tau_j, T_j(k,r)\}\,.$$

Corollary 7. For any (k,r) with $T(k,r-1) < \tau_k$,

$$T_{\text{UBS}}(k,r) \leq T(k,r) \max\{j : T(j,1) \leq T(k,r)\}\,.$$

Example 8. A good choice is $T(k,r) = r2^k$. Then Corollary 7 implies that $T_{\text{UBS}}(k,r) \leq r2^k \lfloor k + \log_2(r) \rfloor$.

8 DovIBEX: Limited Feedback

DovIBEX uses UBS to dovetail multiple instances of exponential search (see Algorithm 5). The algorithm can use any of the three queries, while still exploiting the additional information provided in the extended and integer feedback models. Following Example 8, we use $T(k,r) = r2^k$ so that $T(k,r) - T(k,r-1) = 2^k$. Program k executes one iteration of the loop of exponential search with budget $b_k = 2^k$; if the budget is not entirely used during a segment, the segment ends early. In the limited feedback model, exponential search may continue halving the interval $[C_{\texttt{low}}, C_{\texttt{high}}]$ indefinitely and never halt. The scheduler solves this issue: by interleaving multiple instantiations of exponential search with increasing budgets, the total time can be bounded as a function of the time required by the first program k that finds a solution.

Theorem 9. For $\texttt{query} \in \{\texttt{query}_{\text{ext}}, \texttt{query}_{\text{int}}, \texttt{query}_{\text{lim}}\}$, the number of expansions made by Algorithm 5 is at most

$$Z \lfloor \log_2 Z \rfloor\;, \qquad Z = 2n_* n_{\texttt{exp}}(C_{\min}, C^*, \delta(C^*))$$

and also at most $O(Z' \log Z')$ with

$$Z' = \min_{b \geq n_*} b\, n_{\texttt{exp}}(C_{\min}, C^*, C_{\texttt{crit}}(b) - C^*)\,. \qquad (1)$$

The dependency on $\delta(C^*)$ is gentler than the dependency on δ_{min} in Theorem 3. It means that the behaviour of DovIBEX only depends on the structure of the search space at the solution rather than on the worst case of all iterations.

Remark 10. Sometimes there exists a $b > n_*$ for which

$$b n_{\texttt{exp}}(C_{\min}, C^*, C_{\texttt{crit}}(b) - C^*) \ll n_* n_{\texttt{exp}}(C_{\min}, C^*, \delta_{\text{min}})\,.$$

In these cases the combination of UBS and exponential search can improve on Algorithm 3. Furthermore when using extended feedback, programs $k < k_*$ will now halt if they can prove they cannot find a solution. This allows us to provide the following complementary bound, which is only an additive $2n_* r_2 \lfloor \log_2 r_2 \rfloor$ term away from Theorem 3.

Theorem 11. When $\texttt{query} = \texttt{query}_{\text{ext}}$, the number of expansions made by Algorithm 5 is at most $2n_*(r_1 + r_2(1 + \lfloor \log_2 r_2 \rfloor)$ with $r_1 = n_{\texttt{exp}}(C_{\min}, C^*, \delta_{\text{min}})$, $r_2 = n_{\texttt{exp}}(C_{\min}, C^*, \delta(C^*))$.

Algorithm 5 Dovetailing IBEX

```
1  def run_prog(k, b):
2    [C_low, C_high] = get_state(k, default = [C_min, ∞]
       )
3    if C_high = ∞: # exponential phase
4      C = 2 × C_low
5    else:  # binary phase
6      C = (C_low + C_high)/2
7    [C_low, C_high] = [C_low, C_high] ∩ query(C, b)
8    if C_low >= C_high: return "halted"
9    store_state(k, [C_high, C_low])
10 def DovIBEX():
11   UBS((k, r) ↦ r2^k, run_prog)
```

9 Enhancements

We now describe three enhancements for Algorithm 3 (see Algorithm 6). The enhancements use a modified query function called $\texttt{query}^+_{\text{ext}}$ that returns the same interval as $\texttt{query}_{\text{ext}}$ and the number of expansions, which is $n_{\texttt{used}} = \min\{b, n(C)\}$. For notational simplicity we write $[x, y], n_{\texttt{used}} = \texttt{query}^+_{\text{ext}}(C, b)$.

First, in each iteration of the enhanced IBEX a query is performed with an infinite expansion budget and minimum cost limit $C_{\texttt{low}}$ (Line 7). If the resulting number of expansions is at least $2b$, where b is the current budget, then IBEX updates $C_{\texttt{low}}$, skips the exponential search and moves directly to the next iteration. If furthermore the DFS algorithm is given the lower bound C_k and throws a solution if its cost is C_k, then this guarantees that IBEX performs exactly like IDA* in domains in which the number of expansions grows by at least a factor 2 in each iteration. If the queries with infinite budget (Line 7) do not help skipping iterations, in the worst case they cost an additive $2n_*$ expansions.

The second enhancement is an early stopping condition for proceeding to the next iteration. Algorithm 3 terminates an iteration once it finds $C_{\texttt{crit}}(b)$, which can be slow when $\delta(C_{\texttt{crit}}(b))$ is small. Algorithm 6 uses a budget window defined by $2b$ and αb, where $\alpha \geq 2$, so that whenever a query is made and the number of expansions is in the interval $[2b, \alpha b]$, the algorithm moves on to the next iteration (line 19). Hence iteration k ends when a query is made within budget with a cost within $[C_{\texttt{crit}}(2b), C_{\texttt{crit}}(\alpha b)]$.

The third enhancement is the option of using an additive variant of the exponential search algorithm, which increases $C_{\texttt{low}}$ by increments of 2^j at iteration j during the exponential phase (line 12). This variant is based on the assumption that costs increase linearly when the budget doubles, which often happens in heuristic search if the search space grows exponentially with the depth.

These enhancements can also be applied to DovIBEX (see Algorithm 8 in Appendix C).

Algorithm 6 Enhanced IBEX

```
1  # α: factor on budget, which must be ≥ 2
2  # is_additive: True is using additive search
3  def IBEX_enhanced(α=8, is_additive):
4    C_low = C_min
5    b = 1
6    for k = 1, 2, ...:
7      [C_low, C_high], n_used = query+_ext(C_low, ∞)
8      if n_used < 2b:
9        for j = 1, 2, ...:
10         if C_high == ∞: # exponential phase
11           if is_additive:
12             C = C_low + 2^j
13           else:
14             C = C_low × 2
15         else:  # binary phase
16           C = (C_low + C_high)/2
17         [C'_low, C'_high], n_used = query+_ext(C, α × b)
18         [C_low, C_high] = [C_low, C_high] ∩ [C'_low, C'_high]
19         if (C'_high == ∞ and n_used ≥ 2b) or
20             C_low == C_high:
21           break
22     b = max(2b, n_used)
```

10 Experiments

We test[1] IBEX (BTS, enhanced), DovIBEX (DovBTS, enhanced), IDA* [Korf, 1985], IDA*_CR [Sarkar *et al.*, 1991] and EDA* [Sharon *et al.*, 2014]. EDA*(γ) is a variant of IDA* designed for polynomial domains that repeatedly calls DFS with unlimited budget and a cost threshold of γ^k at iteration k. In our experiments we take $\gamma \in \{2, 1.01\}$. IDA*_CR behaves similarly, but adapts the next cost threshold by collecting the costs of the nodes in the fringe into buckets and selecting the first cost that is likely to expand at least b^k nodes in the next iteration. Our implementation uses 50 buckets and sets $b = 2$. The number of nodes (states) of cost strictly below C^* is reported as $n^{<}_{T*}$ ($n^{<}_{G*}$).

These algorithms are tested for tree search on the 15-Puzzle [Doran and Michie, 1966] with the Manhattan distance heuristic with unit costs and with varied edge costs of $1 + 1/(t + 1)$ to move tile t, on (12, 4)-TopSpin [Lammertink, 1989] with random action costs between 40 and 60 and the max of 3 4-tile pattern database heuristics, on long chains (branching factor of 1 and unit edge costs, solution depth in $[1..10\,000]$), and on a novel domain, which we explain next.

In order to evaluate the robustness of the search algorithms, we introduce the Coconut problem, which is a domain with varied branching factor and small solution density. The heuristic is 0 everywhere, except at the root where $h = 1$. At each node there are 3 'actions', $\{1, 2, 3\}$. The solution path follows the same action (sampled uniformly in $[1..3]$) for D steps, then it follows a random path sampled uniformly

[1] All these algorithms are implemented in C++ in the publicly available HOG2 repository, https://github.com/nathansttt/hog2/.

Algorithm	α	add?	15-Puzzle (unit) Solved	15-Puzzle (unit) Exp.	15-Puzzle (real) Solved	15-Puzzle (real) Exp.	(12, 4)-Topspin Solved	(12, 4)-Topspin Exp.$\times 10^3$	Chain Solved	Chain Exp.$\times 10^4$	Coconut Solved	Coconut Exp.$\times 10^4$
BTS	2	y	100	**242.5**	97	3 214.1	100	1 521.9	100	302.0	100	72.9
	8	y	100	**242.5**	100	**673.1**	100	597.0	100	198.2	100	84.7
	2	n	100	**242.5**	97	3 549.1	100	1 600.0	100	111.8	100	**58.5**
	8	n	100	**242.5**	99	1 320.3	100	614.6	100	26.7	100	86.8
DovBTS	2	y	100	390.5	100	1 087.7	100	1 083.1	100	287.2	100	107.4
	8	y	100	322.0	100	767.4	100	606.1	100	125.9	100	1 136.0
	2	n	100	322.0	98	2 355.2	100	1 145.1	100	33.8	100	121.8
	8	n	100	474.6	100	2 432.2	100	590.5	100	24.9	100	2 882.9
EDA* $\gamma = 2$			99	5 586.0	100	2 882.3	100	807.5	100	**19.4**	3	$\geq$ 554 249.0
EDA* $\gamma = 1.01$			100	1 023.8	100	742.0	100	730.2	100	990.2	10	$\geq$ 528 137.9
IDA*_CR			100	868.4	100	700.6	100	**346.0**	100	988.3	3	$\geq$ 516 937.7
IDA*			100	**242.5**	57	62 044.3	100	2 727.5	100	162 129.0	100	5 484.2
$n^{<}_{T*}$			100	100.8	100	258.1	100	35.8	100	4.9	100	2.7

Table 2: Results on tree search domains. Each tasks has 100 instances. Expansions (Exp.) are averaged on solved tasks only (except Coconut), and times 10^6 for the 15-puzzle. BTS is the implementation for tree search of the IBEX framework. add? is the `is_additive` parameter.

Algorithm	α	add?	d=100 Exp.	d=1 000 Exp.	d=10 000 Exp.	d=10 000 Time (s)
BGS	2	y	2 592	35 478	752 392	0.4
	8	y	1 276	22 275	312 497	0.2
	2	n	2 429	26 030	513 573	0.4
	8	n	513	8 821	84 434	0.1
DovBGS	2	y	2 195	31 862	564 720	0.1
	8	y	1 495	15 757	189 883	0.1
	2	n	1 547	12 987	185 500	0.1
	8	n	**449**	**4 017**	**36 093**	0.1
A*/B/B'			7 652	751 502	75 015 002	22.4
$n^{<}_{G*}$			200	2 000	20 000	0.0

Table 3: Results for inconsistent heuristics in graph search. BGS is the implementation of the IBEX framework for graph search.

of length q, where D is sampled uniformly in $[1..10\,000]$ and q is sampled from a geometric distribution of parameter $1/4$. The first action costs 1. At depth less than D, taking the same action as at the parent node costs 1, taking another action costs $2D$. At depth larger than D, each action costs $1/10$.

10.1 Results

We use 100 instances for each problem domain, a time limit of 4 hours for 15-Puzzle and TopSpin, 1 hour for the Coconut problem and no limit for the Chain problem. The results are shown in Table 2. We report results also for $\alpha = 2$ to show the gain in efficiency when using a budget factor window of $[2, 8]$ instead of the narrower window $[2, 2]$ (see Section 9). IBEX (BTS) and DovIBEX (DovBTS) ($\alpha = 8$) are the only robust algorithms across all domains while being competitive on all domains, whereas all other algorithms tested fail hard on at least one domain. IBEX (BTS) has exactly the same behaviour as IDA* when the number of expansions grows at least by a factor 2 at each call to `query` with infinite budget; see 15-Puzzle (unit). Taking is_additive=y helps on exponential domains, whereas is_additive=n helps on polynomial domains, as expected.

To explain the behaviour of IDA*_CR and EDA* on the Coconut problem, consider a randomly chosen instance where $D = 2\,690$ and $q = 6$. The cost set by EDA* ($\gamma = 2$) in the last iteration is 4 096, resulting in a search tree with approximately $3^{(4\,096-2\,690)/0.1} \approx 10^{6\,700}$ nodes. The same issue arises for IDA*_CR. EDA*(1.01) performs only marginally better. This is not a carefully selected example, and such behaviour occurs on almost all Coconut instances.

Finally, we evaluate our algorithms in graph search problems with inconsistent heuristics, parameterizing Mérõ's (1984) graph by d to have $2d + 2$ states (see Fig. 4 in Appendix D). All states have heuristic of 0 except each state t_i which has heuristic $d + i - 1$. A*, B [Martelli, 1977], and B' [Mérõ, 1984] are all expected to perform $O(n^2_{G*})$ expansions on this graph. The results are in Table 3. While A* shows quadratic growth on the number of expansions, our algorithms exhibit near-linear performance as expected.

11 Conclusion

We have developed a new framework called IBEX that combines exponential search with an increasing node expansion budget to resolve two long-standing problems in heuristic search. The resulting algorithms for tree and graph search improve existing guarantees on the number of expansions from $\Omega(n^2_*)$ to $O(n_* \log C^*)$. Our algorithms are fast and practical. They significantly outperform existing baselines in known failure cases while being at least as good, if not better, on traditional domains; hence, for tree search we recommend using our algorithms instead of IDA*. On graph search problems our algorithms outperform A*, B and B' when the heuristic is inconsistent, and pay only a small $\log C^*$ factor otherwise.

We also expect the IBEX framework to be able to tackle re-expansions problems of other algorithms, such as Best-First Levin Tree Search [Orseau *et al.*, 2018] and Weighted A* [Chen *et al.*, 2019].

The IBEX framework and algorithms have potential applications beyond search in domains that exhibit a dependency between a parameter and the amount of work (computation steps, energy, etc.) required to either succeed or fail. Some of these applications may not be well suited

to the extended feedback model, which further justifies the interest in the analysis of the limited feedback model.

Acknowledgements

This research was enabled in part by a sabbatical grant from the University of Denver and by Compute Canada (www.computecanada.ca). Many thanks to Csaba Szepesvári, András György, János Kramár, Roshan Shariff, Ariel Felner, and the reviewers for their feedback.

References

[Akagi *et al.*, 2010] Yuima Akagi, Akihiro Kishimoto, and Alex Fukunaga. On transposition tables for single-agent search and planning: Summary of results. In *Proceedings of the 3rd Annual Symposium on Combinatorial Search (SoCS 2010)*, pages 2–9, 2010.

[Bentley and Yao, 1976] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976.

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

[Chen *et al.*, 2019] Jingwei Chen, Nathan R. Sturtevant, William Doyle, and Wheeler Ruml. Revisiting suboptimal search. In *Proceedings of the 12th Annual Symposium on Combinatorial Search (SoCS 2019)*, 2019.

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

[Doran and Michie, 1966] James E. Doran and Donald Michie. Experiments with the graph traverser program. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 294(1437):235–259, 1966.

[Felner *et al.*, 2005] Ariel Felner, Uzi Zahavi, Jonathan Schaeffer, and Robert C. Holte. Dual lookups in pattern databases. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 103–108, 2005.

[Felner, 2011] Ariel Felner. Position paper: Dijkstra's algorithm versus uniform cost search or a case against dijkstra's algorithm. In *Proceedings of the 4th Annual Symposium on Combinatorial Search (SoCS 2011)*, pages 47–51, 2011.

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

[Hatem *et al.*, 2015] Matthew Hatem, Scott Kiesel, and Wheeler Ruml. Recursive best-first search with bounded overhead. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015)*, pages 1151–1157, 2015.

[Hatem *et al.*, 2018] Matthew Hatem, Ethan Burns, and Wheeler Ruml. Solving large problems with heuristic search: General-purpose parallel external-memory search. *Journal of Artificial Intelligence Research*, 62:233–268, 2018.

[Holte, 2010] Robert C. Holte. Common misconceptions concerning heuristic search. In *Proceedings of the 3rd Annual Symposium on Combinatorial Search (SoCS 2010)*, pages 46–51, 2010.

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

[Korf, 1993] Richard E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.

[Lammertink, 1989] Ferdinand Lammertink. Puzzle or game having token filled track and turntable, October 3 1989. US Patent 4,871,173.

[Luby *et al.*, 1993] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, September 1993.

[Martelli, 1977] Alberto Martelli. On the complexity of admissible search algorithms. *Artificial Intelligence*, 8(1):1–13, 1977.

[Mérõ, 1984] László Mérõ. A heuristic search algorithm with modifiable estimate. *Artificial Intelligence*, 23(1):13–27, 1984.

[Orseau *et al.*, 2018] Laurent Orseau, Levi H. S. Lelis, Tor Lattimore, and Theophane Weber. Single-agent policy tree search with guarantees. In *Proceedings of the 31st Annual Conference on Neural Information Processing Systems (NeurIPS 2018)*, pages 3205–3215, 2018.

[Orseau *et al.*, 2019] Laurent Orseau, Tor Lattimore, and Levi H. S. Lelis. Zooming cautiously: Linear-memory heuristic search with node expansion guarantees. *CoRR*, abs/1906.03242, 2019.

[Press *et al.*, 1992] William H. Press, Saul A. Teukolsky, William Vetterling, and Brian P. Flannery. *Numerical recipes in C*. Cambridge University Press, 1992.

[Russell and Norvig, 2009] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[Sarkar *et al.*, 1991] U. K. Sarkar, P. P. Chakrabarti, S. Ghose, and S. C. De Sarkar. Reducing reexpansions in iterative-deepening search by controlling cutoff bounds. *Artificial Intelligence*, 50(2):207–221, 1991.

[Sharon *et al.*, 2014] Guni Sharon, Ariel Felner, and Nathan R. Sturtevant. Exponential deepening A* for real-time agent-centered search. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI 2014)*, pages 871–877, 2014.

[Sturtevant and Helmert, 2019] Nathan R. Sturtevant and Malte Helmert. Exponential-binary state-space search. *CoRR*, abs/1906.02912, 2019.

[Sturtevant *et al.*, 2008] Nathan R Sturtevant, Zhifu Zhang, Robert Holte, and Jonathan Schaeffer. Using inconsistent heuristics on A* search. In *Proceedings of the AAAI Workshop on Search Techniques in Artificial Intelligence and Robotics*, 2008.

[Wah and Shang, 1994] Benjamin W. Wah and Yi Shang. Comparision and evaluation of a class of IDA* algorithms. *International Journal on Artificial Intelligence Tools*, 3(4):493–524, 1994.

[Zhou and Hansen, 2004] Rong Zhou and Eric A. Hansen. Structured duplicate detection in external-memory graph search. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI 2004)*, pages 683–689, 2004.

A Graph Search Query

The function graph_search is given in Algorithm 7. The actual query function is like Algorithm 1 where the call to DFS is replaced with a call to graph_search . We use nodes instead of paths to stress that each element of the priority queue uses constant memory size. The fringe is the set of generated nodes with cost larger than C.

Algorithm 7 Graph search (simple version)

```
1 def query_ext(C, budget):
2   data.min_fringe = ∞ # will be > C
3   data.max_visited = 0 # will be ≤ C
4   data.expanded = 0 # number of expansions
5   try:
6     graph_search(C, budget, data)
7   catch "budget exceeded":
8     return [0, data.max_visited]
9   return [data.min_fringe, ∞]
10
11 # make_node: parent, state, g_cost -> node
12 def graph_search(C, budget, data):
13   q = make_priority_queue(g) # g-cost ordering
14   insert(q, make_node(∅, s_init, 0)) #g(s_init) = 0
15   visited = {s_init}
16   while not empty(q):
17     node = extract_min(q)
18     s = node.state
19     if s in visited # already visited
20       continue      # at lower g-cost
21     visited += {s}
22     data.max_visited =
23       max(data.max_visited, node.g + h(s))
24     if is_goal(s): # optimal solution found to
25       throw node # be dealt with by the user
26     if data.expanded >= budget:
27       throw "budget exceeded"
28     data.expanded++
29     for s′ ∈ succ(s):
30       node′ = make_node(node,s′,node.g+c(s,s′))
31       h′ = node′.g + h(s′)
32       if h′ ≤ C:
33         insert(q, node′)
34       else:
35         data.min_fringe =
36           min(data.min_fringe,h′)
37   return "no solution"
```

Enhancement. When the heuristic is sufficiently consistent, A* has an optimal behaviour. In practice, it seldom happens that an inconsistent heuristic leads to a bad behaviour of A*. Therefore, to avoid the (small) overhead of BGS for such cases, we propose the following rule of thumb: Run A* for at least 1000 node expansions. Thereafter, if the number of state re-expansions ever becomes at least half of the total number of expansions, switch to BGS.

B Depth-First Search: Further Enhancements

There are a number of enhancements to DFS that strictly reduce the number of expansions.

- *Upper bounds from sub-optimal solutions.* If a solution is found but the budget is exceeded, then keep the solution cost as an upper bound for subsequent calls.
- Detection of duplicate states can be performed along the current trajectory to avoid loops in the underlying graph, while keeping a memory that grows only linearly with the depth of the search, but is now a multiple of the state size.

C Enhanced DovIBEX Algorithm

An enhanced version of Algorithm 5 is provided in Algorithm 8. The call $\texttt{query}^+(C, b)$ is like $\texttt{query}(C, b)$ but additionally returns the number of node expansions. The optimized version of the algorithm removes programs k from the scheduler if it can be proven that they cannot find a solution. This pruning happens on line Line 20, which condition can be fulfilled in several circumstances.

Let k be the current program index, then if a program $k' > k$ has already made a call to query for which the budget was sufficient and the number of nodes expanded was at least $B(k)$, then program k can never find a solution and is thus removed from the scheduler. Observe that if for program k we have $C_{\texttt{high}} \leq C_{\texttt{low}}$, then we necessarily have $b < b_{\texttt{low}}$, since $b < n(C_{\texttt{high}}) \leq C_{\texttt{low}} = b_{\texttt{low}}$.

On Line 26 we know that $C_{\texttt{low}}$ is a lower bound on the cost of the solution, so it is safe to use infinite budget. This may lead to further pruning on Line 20.

As for IBEX, Algorithm 8 also has two parameters. The first one controls the budget, the second one whether we use an additive exponential search, or a multiplicative one.

With $B(k) = \alpha^k$ and $T(k, r) = r2^k$, it can be worked out that $T_{\text{UBS}}(k, r) \leq \frac{2}{\alpha-2}\alpha^k r^{\log_2 \alpha}$, where $k = \lceil \log_\alpha n_* \rceil$ and $r = r_2$ as in Theorem 9, leading to a number of expansions bounded by $\frac{2\alpha}{\alpha-2} n_* r_2^{\log_2 \alpha}$. This is more efficient than the default setting when r is small, but becomes rapidly less efficient for larger r. But note that this can be mitigated by Eq. (1) and possibly by Theorem 11.

Further enhancements can be considered:

- If a program k has found an upper bound on the cost, then this bound can be propagated to all $k' < k$.
- If a solution has been found but the budget is exceeded, keep the cost of the solution as a global upper bound (global branch and bound).

D Worst-Case Re-Expansions for B and B'

Figure 4 is a parameterized adaptation of an example from [Mérõ, 1984] published by [Sturtevant *et al.*, 2008] where A*, B and B' all perform $O(d^2)$ expansions. Note that [Sturtevant *et al.*, 2008] also introduce the Delay algorithm, however this algorithm has a hidden assumption which is not necessarily true—namely that any state not counted as part of n_{G*} does not have a shorter path to a state counted as part of

Algorithm 8 The Iterative Budgeted Exponential Search algorithm with a few enhancements

```
1  # T: cost function for UBS
2  def T(k, r, C): return r2^k
3
4  # α: Budget parameter
5  # is_additive: use additive exp-search?
6  def DovIBEX_enhanced(α = 8, is_additive):
7    # b_low and C_low are globals, but not C_high
8    C_low = C_min # lower bound on C*
9    b_low = 0 # lower bound on n_*
10   q = make_priority_queue(T)
11   q.insert((1, 1, ∞)) # (k=1, r=1, C_high = ∞)
12
13   while not q.empty():
14     (k, r, C_high) = q.extract_min()
15     b = α^k # budget
16
17     if r == 1:
18       q.insert((k+1, r, ∞))
19
20     if b ≤ b_low or C_high ≤ C_low:
21       # Can't find a solution with this budget
22       continue  # remove k from the queue
23
24     if r == 1:  # IDA* trick, can be omitted
25       C = C_low
26       b = ∞
27     else if C_high == ∞: # exponential phase
28       if is_additive:
29         C = C_low + 2^(r−1)
30       else:
31         C = C_low × 2
32     else: # binary phase
33       C = (C_low + C_high)/2
34
35     C'_low, C'_high, b'_low = query+(C, b)
36     [C_low, C_high] = [C_low, C_high] ∩ [C'_low, C'_high]
37
38     if C'_high == ∞: # budget not exceeded
39       # Solution requires more than b'_low
40       b_low = b'_low
41
42     q.insert((k, r+1, C_high))
```

n_{G*}. As this is not true in general, the Delay algorithm is not completely general.

E Intractability of Admissible Heuristics

Let $c^*(s', s)$ be the minimum cost of any path from s' to s with $c^*(s, s) = 0$. Assuming that the heuristic h is admissible, define

$$\hat{h}(s) = \max_{s' \in \mathcal{S}} \{h(s') - c^*(s', s)\}\,,$$

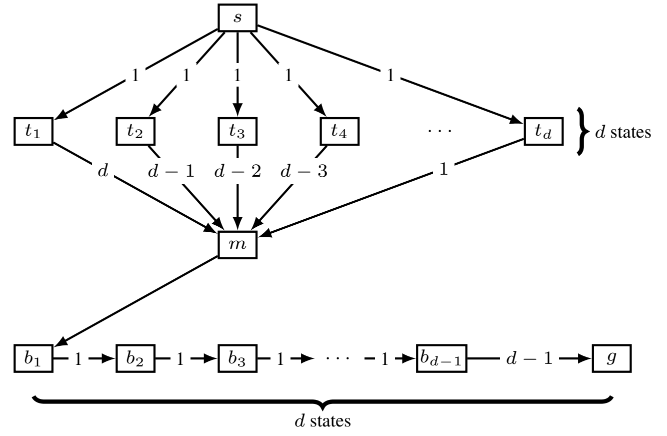

Figure 4: Worst-case example adapted from [Mérõ, 1984] by [Sturtevant *et al.*, 2008].

which is the best heuristic value that can be propagated forward to s. Note that $\hat{h}$ is consistent.

Let C^* be minimum cost of any solution state. Let $n_{\text{opt}} = |\{s : g^*(s) + \hat{h}(s) \le C^*\}|$, which is an upper bound on the number of states that A* expands when using $\hat{h}$ instead of h. The following theorem shows that no algorithm having only access to h can hope to expand $O(n_{\text{opt}})$ without oracle access to the consistent heuristic $\hat{h}$.

Theorem 12. For each deterministic search algorithm there exists a graph search problem with an admissible heuristic such that the algorithm expands $\Omega(2^{n_{\text{opt}}})$ nodes.

Proof. Let d be an arbitrarily large integer. Consider the following class of problems, illustrated in Fig. 5. The initial state is s_{init}, which has $g(s_{\text{init}}) = 0$ and successors $\texttt{succ}(s_{\text{init}}) = \{s_1, s_2\}$. Below s_2 is a full binary tree of depth d. State s_1 has successors $\texttt{succ}(s_1) = \{l_1, r_1, \ldots, l_d, r_d\}$, each of which starts a chain of length 2^d with the final state in the chains started by l_m (respectively r_m) connected to all left-hand (respectively right-hand) children in the binary tree at depth m. The heuristic values are 0 except $h(s_1) = 2^d + d + 3$. Consider the first $2^d - 1$ expansions of the search algorithm where all edge costs are unitary and there is no goal state. By the pigeonhole principle there exists a state s in the leaves of the binary tree that has not been expanded. We now modify the graph so that (a) the behavior of the algorithm is identical (b) the goal is in state s^* and (c) $n_{\text{opt}} = d + 2$. To do this, let s^* be the goal state and $(a_t)_{t=1}^{d+2}$ be the unique path ending in s^* and passing through s_2. Then for each $3 \le m \le d + 2$ find an edge in the chain connected to state a_m that was not examined by the algorithm in the goal-less graph and set its cost to infinity, which cuts the path from state s_1 to state s_m. Since the algorithm has not examined this edge, it cannot prove whether the heuristic value of s_1 should be propagated to the corresponding node in the full binary tree. The large heuristic in state s_1 ensures that children in the binary tree that do not lead to the goal state have a heuristic of at least $d+3$, whereas all states along the path a have heuristic 0. Hence $n_{\text{opt}} = d+2$. Finally, by construction the algorithm

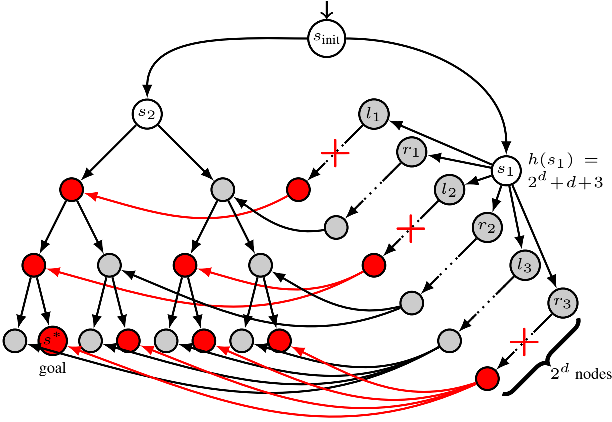

Figure 5: The graph for the proof of Theorem 12 with $d = 3$. The large heuristic value of s_1 propagates through all the gray nodes. Red crosses in the chains indicate that the chain contains an edge with infinite cost, preventing the large heuristic from being propagated to the full binary tree. Only one path in the tree is entirely red, and leads to the goal g.

expands at least $2^d - 1$ nodes in the modified graph before finding the goal. The result follows since d may be chosen arbitrarily large. □

Remark 13. The example is complicated by the fact that we did not assume the algorithm was restricted to only call the successor function on previously expanded nodes. We only used that an edge-cost is observed when its parent is expanded. We did not even assume that the algorithm is guaranteed to return an optimal solution. The proof is easily modified to lower bound the expected number of expansions by any randomized algorithm using Yao's minimax principle and by randomizing the position of the goal in the leaves of the binary tree and the infinite-cost edges in the chains.

Remark 14. Theorem 12 also holds when using pathmax [Mérõ, 1984] or BPMX [Felner *et al.*, 2005], since in order to make the heuristic consistent the algorithm still needs to expand exponentially many nodes along the chains.

F Proofs of Propositions 1 and 2

In the following we use $\varepsilon = \texttt{start}$, $h = \texttt{high}$ and $l = \texttt{low}$.

Proof of Proposition 1. We know that $C^* < C_{\texttt{crit}}(b)$ by definition of b through k, and that $\varepsilon \in [C^*, C_{\texttt{crit}}(b))$.

The number of queries in the exponential phase before $h \geq C^*$ is at most

$$\min\{k \in \mathbb{N}_1 : \varepsilon 2^k \geq C^*\} = \left\lceil \log_2 \frac{C^*}{\varepsilon} \right\rceil_{\geq 1} .$$

At the end of the exponential phase, if $h \in [C^*, C_{\texttt{crit}}(b))$ then the game ends. Otherwise $h \geq C_{\texttt{crit}}(b)$, and also $l \leq C^*$ (otherwise a game-ending query would have been made; $l = C^*$ is possible iff $\varepsilon = C^*$). Thus $l \leq C^* < C_{\texttt{crit}}(b) \leq h$, and since $h = 2l$ we have $h - l \leq C^*$. During the binary phase, the size of the interval $[l, h]$ at least halves after each query. Now, suppose that at some point $h - l < 2(C_{\texttt{crit}}(b) - C^*)$. Using $C = (h + l)/2$ then we obtain $C < l + C_{\texttt{crit}}(b) - C^* \leq C_{\texttt{crit}}(b)$ since $l \leq C^*$, and also $C > h - (C_{\texttt{crit}}(b) - C^*) \geq C^*$ since $h \geq C_{\texttt{crit}}(b)$. Therefore $C \in (C^*, C_{\texttt{crit}}(b))$ which is a game-ending query. Starting from $h - l \leq C^*$, this requires at most

$$\begin{aligned} &1 + \min\{k \in \mathbb{N}_0 : C^*/2^k \leq 2(C_{\texttt{crit}}(b) - C^*)\} \\ &= 1 + \left\lfloor 1 + \log_2 \frac{C^*}{2(C_{\texttt{crit}}(b) - C^*)} \right\rfloor_{\geq 0} \\ &= 1 + \left\lfloor \log_2 \frac{C^*}{C_{\texttt{crit}}(b) - C^*} \right\rfloor_{\geq 0} \end{aligned}$$

calls to query before ending the game. Therefore the number of calls to query is at most $n_{\text{exp}}(\texttt{start}, C^*, C_{\texttt{crit}}(b) - C^*)$. □

Proof of Proposition 2. Let $x < y < z$ be three consecutive values in A (that is, $y = A_>(x)$ and $z = A_>(y)$) with $y = C_{\texttt{crit}}(b)$. During the exponential phase, the number of queries until $h \geq C_{\texttt{crit}}(b)$ is at most

$$\begin{aligned} \min\{k \in \mathbb{N}_1 : \varepsilon 2^k \geq C_{\texttt{crit}}(b)\} &= \left\lceil \log_2 \frac{C_{\texttt{crit}}(b)}{\varepsilon} \right\rceil_{\geq 1} \\ &\leq \left\lceil \log_2 \frac{C^*}{\varepsilon} \right\rceil_{\geq 1} . \end{aligned}$$

At the end of this phase, we have $l \leq C_{\texttt{crit}}(b) \leq h$ and since $h = 2l$ we have $h - l \leq C_{\texttt{crit}}(b)$. Now for the binary phase, where the interval $[l, h]$ is always at least halved. Observe that $\texttt{query}_{\text{ext}}$ ensures that if $x \leq C < y$ then l is set to y, and if $y \leq C < z$ then h is set to y too. Next we show that if $h - l < \delta_{\text{min}}$ then $h < z$ and $l > x$:

$$\begin{aligned} h - l &< \delta_{\text{min}} \leq z - y && \text{(by assumption)} \\ h &< z - y + l \leq z && \text{(using } l \leq y\text{)} \\ h - l &< \delta_{\text{min}} \leq y - x && \text{(by assumption)} \\ l &> x - y + h \geq x\,. && \text{(using } h \geq y\text{)} \end{aligned}$$

(Remembering that $y = C_{\texttt{crit}}(b)$, observe that if $\texttt{start} = y$ then $l = y$ already at the beginning of Algorithm 2.) Thus $h = l = y$ which terminates the algorithm and returns $l = C_{\texttt{crit}}(b)$.

Hence the number of calls to query during the binary phase before $h - l < \delta_{\text{min}}$ (which entails $h = l$) is at most (remembering that $h - l \leq C_{\texttt{crit}}(b)$ at the end of the exponential phase)

$$\begin{aligned} &\min\{k \in \mathbb{N}_0 : C_{\texttt{crit}}(b)/2^k < \delta_{\text{min}}\} \\ &= \left\lfloor 1 + \log_2 \frac{C_{\texttt{crit}}(b)}{\delta_{\text{min}}} \right\rfloor_{\geq 0} \\ &\leq 1 + \left\lfloor \log_2 \frac{C^*}{\delta_{\text{min}}} \right\rfloor_{\geq 0} . \end{aligned}$$

Therefore, the total number of queries is at most $n_{\text{exp}}(\texttt{start}, C_{\texttt{crit}}(b), \delta_{\text{min}})$. □

G Proofs of Theorem 6 and Corollary 7

Proof of Theorem 6. Suppose when UBS is executing (k, r), the node (j, m) has already been executed by UBS. Then the optimality property of Uniform Cost Search [Russell and Norvig, 2009] ensures that $T(j, m) \leq T(k, r)$. Hence, program j has been executed for at most $T_j(k, r)$ steps. The result follows by summing over all programs and using that program j never runs for more than τ_j steps. □

Proof of Corollary 7. The result follows from Theorem 6 and the assumptions that $T(j, 0) = 0$ and $T(j, 1) \leq T(j, m)$ for $m \geq 1$. □

H Proofs of Theorems 9 and 11

Proof of Theorem 9. Let $k^* = \lceil \log_2(n_*) \rceil$. By Proposition 1, programs with $k \geq k^*$ make a game-ending query after at most

$$r_k = n_{\texttt{exp}}(C_{\min},\ C^*,\ C_{\texttt{crit}}(b_k) - C^*)$$

calls to run_prog $(k, 2^k)$. By Theorem 6, Algorithm 5 does not use more than $\min_{k \geq k^*} r_k 2^k \lfloor k + \log_2 r_k \rfloor$ steps before exiting with a solution, which also upper bounds the number of expansions of the algorithm. The first bound is obtained by taking $k = k^*$ for which $b_{k^*} \leq 2n_*$ and $C_{\texttt{crit}}(2^{k_*}) \geq C_{\texttt{crit}}(n_*) = C^* + \delta(C^*)$. The second bound is obtained by noting that for any $b \geq n_*$ there exists a k such that $2^k \geq n_*$ and $r_k 2^k \leq 2bn_{\texttt{exp}}(C_{\min}, C^*, C_{\texttt{crit}(b)} - C^*)$. □

Proof of Theorem 11. Similarly to the proof of Theorem 3, let $k_* = \lceil \log_2(n_*) \rceil$ be the first program with enough budget, that is $2^{k_*} \geq n_*$. Using Proposition 1, program k_* terminates after at most r_2 calls to `query`. Each program $k < k_*$ requires at most r_1 calls to `query` to terminate, that is $\tau_k \leq 2^k r_1$. Each program $k > k_*$ that has started when k_* terminates has used at most $T_k(k_*, r_2)$ steps, and only programs $k \leq k_* + \lfloor \log_2 r_2 \rfloor$ (that is $2^k \leq r_2 2^{k_*}$) have started (that is, $T_k(k_*, r_2) > 0$). Hence, using Theorem 6, the number of steps is bounded by

$$\begin{aligned}
T_{\text{UBS}}(k_*, r_2) &\leq \sum_{k < k_*} \tau_k + \sum_{k \geq k_*} T_k(k_*, r_2) \\
&\leq \sum_{k=1}^{k_*-1} 2^k r_1 + \sum_{k=k_*}^{k_* + \lfloor \log_2 n_2 \rfloor} 2^{k_*} r_2 \\
&= (2^{k_*} - 1) r_1 + (1 + \lfloor \log_2 r_2 \rfloor) 2^{k_*} r_2 \\
&\leq 2n_*(r_1 + r_2(1 + \lfloor \log_2 r_2 \rfloor)),
\end{aligned}$$

which also bounds the number of expansions of the algorithm. □