A Memory-Bounded Best-First Beam Search and Its Application to Scheduling Halide Programs

Chao Gao,[1] Jingwei Chen,[1] Tong Mo,[1] Tanvir Sajed,[1] Shangling Jui,[2] Min Qin,[2] Laiyuan Gong,[2] Wei Lu[1]

[1]Huawei Canada Research Center
[2]Huawei Technologies, China
{chao.gao4, tong.mo, jingwei.chen, tanvir.sajed, jui.shangling}@huawei.com
{gonglaiyuan,qinmin5,robin.luwei}@hisilicon.com

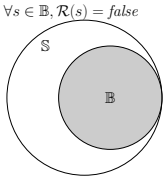
Abstract

Beam search is a popular algorithm for solving real-world problems — especially where search space is an enormously large tree but real-time solutions are most preferred. We present a memory-bounded best-first beam search (MB2FBS), which can be viewed as an improved and generalized version of standard beam search in trees. The algorithm takes three parameters — in contrast to the singular parameter beam size in standard beam search. We discuss how to recover standard beam search and how to realize other search behaviour by setting these three parameters correspondingly. In particular, we show that the principal version of MB2FBS can be thought as an algorithm whose search expense is similar or upper bounded by beam search of certain beam size; however it often finds better solutions as it decides the number of nodes to be searched each depth dynamically with respect cost landscape. We apply our algorithm for tensor program auto-scheduling in Halide, an important industrial problem that uses tree search for optimizing tensor program executions. We show that the principal variants of MB2FBS deliver better empirical results than the highly optimized beam search counterpart. Most importantly, it finds superior schedules while no more computation cost is used for search, which is highly desirable for real-time program compilation and optimization.

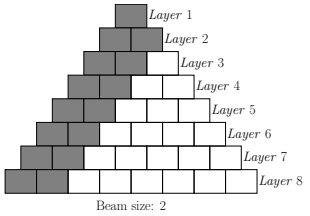
Introduction

Beam search Medress et al. is a heuristic search paradigm that has been extensively used in solving problems from many areas of artificial intelligence, such as vision (Roy and Todorovic 2014), job scheduling (Birgin, Ferreira, and Ronconi 2015), planning (Karbowska-Chilinska et al. 2019) and natural language processing (Meister, Vieira, and Cotterell 2020) — in these domains, beam search is preferred presumably because the computer memory is typically insufficient for searching the whole state space of a problem instance, meanwhile a quick and sub-optimal solution is more desirable than an optimal but slow one.

Essentially, beam search achieves fast solving by searching only on a promising subset of nodes from the whole state-space $\mathbb{S}$, assuming there is an external input for heuristic pruning. Indeed, there have been two popular ways of characterizing beam search (Bisiani 1992): the general form describes beam search as an abstract algorithm where search only has to be conducted on a subset of states $\mathbb{B}$ (called *beam*) induced by a set of heuristic pruning rule $\mathcal{R}$ — the search strategy used to explore $\mathbb{B}$ can either be depth-first, breadth-first or best-first; the another simplified and more specific form of beam search regards the algorithm as a *truncated* version of *breadth-first search* — given a beam size parameter $b > 0$ and a priority function for comparing states, the search needs only to consider top b nodes each layer. Evidently, the second form can be seen as a specific realization of the first version. However, in practice, the simplified form is more popularly used such that it is often referred as the *standard* version of beam search (Zhou and Hansen 2005). Figure 1 illustrates *general* and *standard* breadth-first based beam search.



(a) General beam search using a set of predefined heuristic pruning rule $\mathcal{R}$. Thus, it searches only nodes in subset $\mathbb{B} \subseteq \mathbb{S}$.



(b) Standard beam search with parameter b. At each search depth, it checks only b nodes.

Figure 1: Illustration of *general* and standard beam search.

The principal drawback of beam search is that the algorithm by design does not guarantee *completeness*. To address this issue while maintaining its advantage of being able to deliver solutions using short time and small computer memory, subsequent algorithms have been developed to embed beam search into another systematic searching such that completeness can be achieved. To name a few, (Zhang

1998) proposed *anytime complete beam search* that iteratively calls beam search with relaxed pruning rules so that convergence to optimal can be assured; (Zhou and Hansen 2005) proposed *beam-stack search* that conducts a sequence of standard beam search using depth-first search; (Furcy and Koenig 2005) devised *limited discrepancy beam search* that performs limited discrepancy search (Harvey and Ginsberg 1995; Korf 1996) upon standard beam search.

On the other hand, standard beam search might be viewed as an extension of *greedy search*. Like greedy search, beam search selects successors to expand consecutively until a solution is reached; unlike greedy search, at each time, a *set of successor states*, rather than a single state, are selected to proceed. This continual *pushing forward* ensures fast execution of beam search; however, we argue that this scheme is also the source of deficiency of standard beam search because that it neglects the comparison of states across different *depths*. Instead of implicitly limiting the memory usage through heuristic pruning rules, another line of research have been pursued by giving an explicit memory-bound to the search. The memory-bounded A* (MA*) algorithms (Chakrabarti et al. 1989; Russell 1992; Kaindl and Khorsand 1994; Zhou and Hansen 2002) work similar to A* (Hart, Nilsson, and Raphael 1968) except that it limits the maximum number of searching nodes in OPEN and CLOSED list, and whenever the memory is full, a worst-seeming node in OPEN is released and its parent would be moved from CLOSED to OPEN if necessary. These algorithms guarantee that the search would be able to find an optimal solution as long as the memory is sufficient to store the least sized optimal solution path. In fact, MA* algorithms sit in between IDA* (Korf 1985)/RBFS (Korf 1993) and A*, in the sense that the former uses the least memory possible at the expense of extensive *re-expansion*, while the later requires a memory as large as the whole state-space but avoids possible re-expansion at its extreme.

Motivated by developing better incomplete but fast algorithms for scheduling Halide programs (Ragan-Kelley et al. 2013; Adams et al. 2019)[1], in this paper, we address the deficiency of beam search using best-first search. We present a generalization of standard beam search from the perspective of memory-bounded and $k-$best-first search (KBFS) (Felner, Kraus, and Korf 2003). We illustrate the usefulness of the new search algorithm with a synthetic tree environment whose cost landscape can be manipulated for specific interests of investigation. Finally, experiments in Halide demonstrate that our new algorithm surpasses the state-of-the-art beam-search auto-scheduler without incurring more computation budgets.

Preliminaries

To ease presentation and analysis, we assume the *state-space* $\mathbb{S}$ *to be searched can be regarded as a tree*, such that the following terminologies and assumptions are used:

- There is a singular root node called *start*.
- $\forall s \in \mathbb{S}$, $ch(s) \subset \mathbb{S}$ represents the set of children below s, while $s \neq start, pa(s) \in \mathbb{S}$ stands for the parent of s.
- A node $s \in \mathbb{S}$ is called *terminal* if $ch(s) = \emptyset$.
- Following A* algorithms, $\forall s \in \mathbb{S}$, $g(s)$ is the path cost $start \rightarrow s$, while $h(s)$ estimates the cost from s to goal. $h(s)$ is called *admissible* if $h(s) \leq h^*(s)$, i.e., $h(s)$ underestimates the optimal cost from s to goal. $f(s) \triangleq g(s) + h(s)$.
- All costs are non-negative.

Starting from the root node, *best-first* search algorithms construct a search tree $\mathbb{T}$ iteratively through a sequence of node expansions. Formally, $\forall s \in \mathbb{T}$, s is called *closed* if $ch(s) \subset \mathbb{T}$ otherwise *open*, and suppose they are respectively stored in sets CLOSED and OPEN, then $\mathbb{T} =$ CLOSED $\cup$ OPEN. At each iteration of best-first search, a node with the lowest (thus *best*) f cost in OPEN is selected for expansion, and when h is admissible everywhere, it is known that the best-first search becomes A* (Pearl 1984). A best-first search is called *memory-bounded* if there is a maximum storage limit for CLOSED $\cup$ OPEN such that whenever the limit is reached, node deletion has to be performed on OPEN. In this paper, we do not store CLOSED, since our major concern is not *completeness* (i.e., backtracking from CLOSED as SMA* (Russell 1992) is not needed).

In standard beam search, the parameter b is called *beam size*; for each search depth, states are sorted in ascending order w.r.t f, such that top b states are selected for searching.

Unified Framework for Memory-Bounded Best-First Beam Search

In this section, we present our new algorithm framework for memory-bounded best-first beam search.

A Simple Memory-Bounded Best-first Search

We first discuss a simple algorithm that executes in the same spirit as best-first search except that a bounded memory is used. The algorithm is sketched in Algorithm 1. The priority queue can be implemented using min-max priority queue (Atkinson et al. 1986), such that whenever the Q is exceeding its bound, a worst state w.r.t ρ can be deleted. A natural choice for ρ is the f cost, as in A*, and it is clear that when Q is given a max size larger than the total number of possible states, Algorithm 1 becomes A*, assuming that the heuristic function h is admissible — in this case, without losing optimality, Algorithm 1 can stop early by returning its first encountered terminal. It is also clear that Algorithm 1 expands $\mathcal{O}(d_{max}M)$ nodes, since at each tree depth, it would never visit more than M nodes due to the memory bound.

Batched Expansion Equals Beam Search

We see that Algorithm 1 updates the bounded memory after every node expansion. We now consider the case that the insertion to Q is delayed until a batch of nodes have been expanded — we refer this scheme as *batched expansion*. That is, in Algorithm 2, we modify the $s \leftarrow Q.pop() \Longrightarrow S_k \leftarrow Q.pop(k)$ where k is batch size, and $C \leftarrow expand(s) \Longrightarrow C \leftarrow expand(S_k)$.

[1] https://halide-lang.org/

Algorithm 1: Simple memory-bounded best-first search for tree

```
1  Procedure SMBS(start, ρ, M):
       /* ρ is priority function such
          that ρ(x) ⊵ ρ(y) implies x has
          higher priority than y        */
2      Q ← priority_queue<ρ>(max_size = M)
3      r ← null
4      while not Q.empty() do
5          s ← Q.pop()
6          if s.terminal() then
7              if r = null or ρ(s) ⊵ ρ(r) then r ← s
8              continue
9          C ← expand(s)
10         foreach c ∈ C do Q.push(c)
11     return r
```

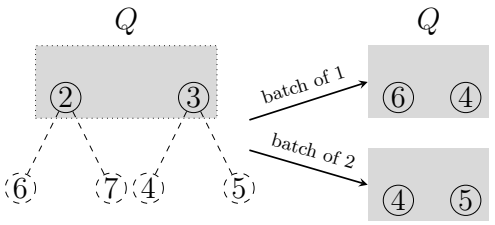


Figure 2: Memory-bounded best-first search batch 2 expansion vs one-by-one expansion. Each number indicates a f-cost, thus lower cost implies higher priority. For batch 1 expansion, first, node ② is expanded, then $Q \leftarrow \{⑥, ③\}$, then ③ gets expanded, $Q \leftarrow \{⑥, ④\}$. For batch 2 expansion, ② and ③ are expanded at the same time, then $Q \leftarrow \{④, ⑤\}$.

To see the effect of batched expansion, consider a case illustrated in Figure 2, where if Q has a bounded size of 2, one by one expansion would update $Q \leftarrow \{④, ⑥\}$ whereas batch of 2 expansion results $Q \leftarrow \{④, ⑤\}$. Thus, we have the following observation.

Lemma 1. *If a batch of k nodes expansion is applied, then, in the bounded memory Q, there must be k states being pushed forward in the next iteration, i.e., these k states in Q will be replaced by one-depth deeper states.*

Theorem 1. *For simple memory-bounded best-first search, if we limit the priority queue size to β, and use a batch expansion of β states each time, then, the algorithm is equivalent to standard beam search.*

MB2FBS: Mixing Push Forward and Onward

While Algorithm 1 makes sure that whenever a batch of k nodes is expanded, there is a *push forward* by k in Q, the deficiency of this scheme is that it neglects the other states at lower depth that was not considered — they may become promising again upon the removal of just expanded states.

Consider a case illustrated in Figure 3. Suppose at certain time the Q contains two nodes of costs 2 and 3, and they have a sibling not in Q with a cost of 4. If we expand a batch of two nodes, then update the Q by the newly created child nodes, then $Q \leftarrow \{⑤, ⑥\}$; however, if we also re-examine the sibling of ② and ③, then $Q \leftarrow \{④, ⑤\}$ — this is arguably a better strategy since nodes with smaller cost estimates are retained. To distinguish, we name this strategy of reconsideration as *pushing onward*.

It is clear that pushing forward guarantees fast execution to terminal states, while pushing onward makes sure that the most promising states are pursued. In order to subsume the advantages of both strategies, we devise a new search algorithm where the extent of each strategy can be swiftly configured by appropriate parameterization. Specifically, we introduce two parameters, β_1 and β_2, respectively represent the number of states being pushed forward and onward at each iteration. The full procedure is depicted in Algorithm 2.

Same as Algorithm 1, Algorithm 2 operates on a bounded memory parameterized by M, which we assume should be larger than or equal to $\beta_1 + \beta_2$. In fact, any $M > \beta_1 + \beta_2$ is equivalent to $M = \beta_1 + \beta_2$ — for each iteration in Algorithm 2, sorting the nodes in Q by their priority, then the first top β_1 nodes from Q are selected for expansion and newly generated child nodes are inserted to U, then next top β_2 nodes from Q are directly moved from Q to U. In other words, even Q contains more than $\beta_1+\beta_2$ nodes, these nodes not in top $\beta_1 + \beta_2$ will simply be ignored. Thus, in practice, setting $M = \beta_1 + \beta_2$ suffices. Assuming $M = \beta_1 + \beta_2$, we can now bound the number of node expansion of Algorithm 2 in Proposition 1.

Proposition 1. *For Algorithm 2 with parameters $\beta_1 > 0, \beta_2 > 0, \beta = \infty$, and $M = \beta_2 + \beta_1$, then the total number of node expansion is bounded by $\mathcal{O}((\beta_1 \lceil \frac{\beta_2}{\beta_1} \rceil) \frac{d_{max}(d_{max}+1)}{2})$*

Proof. The algorithm begins with all states in $\mathcal{Q}$ with depth 0, and ends with all states at maximum depth. In other words, let x be the minimum state depth in Q, then it starts with $x = 0$, and ends with $x = d_{max}$. We know that for each iteration of search, β_1 states are expanded, thus that x either remains unchanged or $x \leftarrow x + 1$. Therefore, the slowest possible growth for x is that the same β_2 states being retained for the longest time.

Since $M = \beta_1 + \beta_2$ and $\beta_1 \geq \beta_2$, then it takes at most d_{max} iterations for the same β_2 states being retained in Q. Formally, denoting the time complexity as $T(d_{max})$, we can express the worst case recursively as follows:

$$
\begin{aligned}
T(d_{max}) &\leq \beta_1 d_{max} + T(d_{max} - 1) && (1)\\
&= \beta_1 d_{max} + \beta_1 (d_{max} - 1) T(d_{max} - 2) && (2)\\
&= \beta_1 (d_{max} + (d_{max} - 1) + \ldots + 1) && (3)\\
&= \beta_1 d_{max}(d_{max} + 1)/2 && (4)\\
& && (5)
\end{aligned}
$$

For he other case that $\beta_1 < \beta_2$ ($M = \beta_1 + \beta_2$), only a

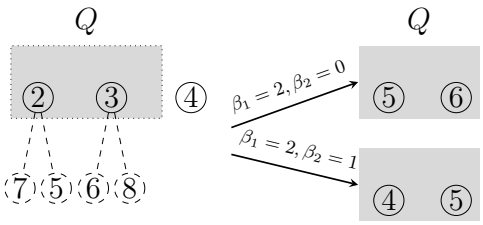


Figure 3: Only push forward versus mixed push forward and onward. For $\beta_1 = 2, \beta_2 = 0$, ② and ③ are expanded the same time, then $Q \leftarrow \{⑤, ⑥\}$ while node ④ is discarded forever. For $\beta_1 = 2, \beta_2 = 1$, ② and ③ are expanded, and ④ is re-examined while updating Q, thus $Q \leftarrow \{④, ⑤\}$

.

factor is needed for the recursion:

$$T(d_{max}) \leq \beta_1 d_{max} \lceil \frac{\beta_2}{\beta_1} \rceil + T(d_{max} - 1) \quad (6)$$

$$= \beta_1 \lceil \frac{\beta_2}{\beta_1} \rceil d_{max}(d_{max} + 1)/2 \quad (7)$$

$$= \mathcal{O}((\beta_1 \lceil \frac{\beta_2}{\beta_1} \rceil) \frac{d_{max}(d_{max} + 1)}{2}). \quad (8)$$

Thus, the claim holds. □

We can draw the following observation from Proposition 1: for a given search depth, multiple batches of β_1 states might be eventually expanded — in contrast to standard beam search, where a constant of b states will be expanded. To address this discrepancy and let Algorithm 2 matches the efficiency of beam search, we then introduce counting scheme to control the number of states being expanded at each depth — we can force the algorithm not to expand more than β ($\beta < \infty$) nodes. We name Algorithm 2 with $\beta < \infty$ as the *controlled* version of MB2FBS. We note that, for the controlled version, M has to be at least βd_{max}, otherwise the algorithm might be unable to return a solution. That is, if Q and U are with too small capacity, new nodes created by batch expanding β_1 nodes might all get discarded, and if this is repeated multiple rounds till $\beta_1 + \ldots + \beta_1 > \beta$, then the algorithm would stop moving forward forever. This phenomenon is also seen in best-first beam search (Meister, Vieira, and Cotterell 2020). In Algorithm 2, we mark how β controls the number of expansion using a vector of $POPS$. For the uncontrolled version, these code with $POPS$ becomes ineffective because $\beta = \infty$.

Remark. *For both uncontrolled and controlled versions of MB2FBS, in practical use, we can skip to set the parameter M (i.e., let Q and U unbounded by assuming $M \leftarrow \infty$), because the memory-usage of Q and U are implicitly implied by the setting of β_1, β_2 and β. If one wants to set M, it must be at least either $\beta_1 + \beta_2$ (uncontrolled) or βd_{max} (controlled).*

It is possible to configure the parameters, i.e., β_1, β_2, β, swiftly for realizing different search behaviors. In Table 1 we summarize how to set these parameters for 8 typical variants.

Algorithm 2: Memory-bounded best-first beam search for tree

```
1  Procedure MB2FBS(start, ρ, β1, β2, M, β = ∞):
       /* ρ is priority function such
          that ρ(x) ⊵ ρ(y) implies x has
          higher priority than y        */
2      Q ← priority_queue<ρ>(max_size = M)
3      POPS ← vector(d_max)
4      r ← null
5      while not Q.empty() do
6          U ← priority_queue<ρ>(max_size = M)
7          for i ∈ {1, 2, ..., β1 + β2} ∧ not Q.empty()
             do
8              s ← Q.pop()
9              if s.terminal() then
10                 if r = null or ρ(s) ⊵ ρ(r) then
                     r ← s
11                 continue
12             if i > β1 then
13                 U.push(s)
14                 continue
15             if POPS[s.depth] ≥ β then continue
16             C ← expand(s)
17             POPS[s.depth] ← POPS[s.depth] + 1
18             foreach c ∈ C do U.push(c)
19         Q ← U
20     return r
21
```

- If M is assumed to be as large as the whole tree, and $\beta_2 = 0, \beta_1 = \beta = \infty$, then Algorithm 2 becomes an instance of breadth-first search where the visit of states in the same layer is prioritized by priority function ρ. If $\beta_1 = 1, \beta_2 = \infty$, then the algorithm becomes an equivalent of best-first search — more generally, KBFS (Felner, Kraus, and Korf 2003), if $\beta_1 > 1$. The best-first search becomes an implementation of A* when the priority function is defined using an admissible heuristic, e.g., $f = g + h$ where h is admissible.
- If $\beta_1 = 1, \beta_2 = 0$, Algorithm 2 becomes greedy search. If $\beta_1 = 1, \beta_2 = \infty$, then the algorithm becomes best-first search except that M is bounded, akin to Algorithm 1.
- Suppose β is given, and we set $\beta_2 = 0, \beta_1 = \beta$, then Algorithm 2 works the same as standard beam search with beam size $b = \beta$. If $\beta_1 = 1, \beta_2 = \beta - 1$, then, the algorithm becomes an equivalent of the best-first beam-search depicted by (Meister, Vieira, and Cotterell 2020). These two algorithms are guaranteed to find the same solution if ρ is defined upon admissible heuristic.
- The last row of Table 1 contains the two variants of our new algorithm MB2FBS. The controlled version ensures that it expands no more than βd_{max} nodes, which is the number of nodes get expanded by beam search with $b = \beta$.

Breadth-first Search $\beta_1 \leftarrow \infty, \beta_2 \leftarrow 0, \beta \leftarrow \infty$ $M = {b_{max}}^{d_{max}}$	Best-first Search $\beta_1 \leftarrow 1, \beta_2 \leftarrow \infty, \beta \leftarrow \infty$ $M = {b_{max}}^{d_{max}}$
Greedy Search $\beta_1 \leftarrow 1, \beta_2 \leftarrow 0, \beta \leftarrow 1$ $M = 1$	Mem. bounded Best-first Search $\beta_1 \leftarrow 1, \beta_2 \leftarrow \infty, \beta \leftarrow \infty$ $M > 1$
Standard Beam Search $\beta_1 \leftarrow \beta, \beta_2 \leftarrow 0$ $M = \beta$	Best-first Beam Search $\beta_1 \leftarrow 1, \beta_2 \leftarrow \beta - 1$ $M = d_{max}\beta$
MB2FBS-β-controlled $\beta_1 > 0, \beta_2 > 0$ $M = d_{max}\beta$	MB2FBS $\beta \leftarrow \infty, \beta_1 > 0, \beta_2 > 0$ $M = \beta_1 + \beta_2$

Table 1: Possible configuration of MB2FBS for other well-known heuristic search algorithms.

For each variant, the corresponding M value in Table 1 indicates the memory requirement of U and Q to ensure each algorithm run correctly. When β_2 is large and β_1 is small (e.g., $\beta_1 = 1, \beta_2 = \infty$), the frequent node movement from Q to U then U to Q in Algorithm 2 might become an efficiency bottleneck. However, this issue can be addressed by modifying the implementation, for example, by popping out the β_1 states from Q to a separate container, then expanding each of them while directly inserting the newly generated child nodes to Q. To keep the pseudocode simple, we ignore specific implementation enhancement for these cases in Algorithm 2.

Experiments

We now proceed to deploy MB2FBS for auto-scheduling in Halide (Ragan-Kelley et al. 2013). We begin by introducing the problem, how the scheduling can be modelled as a tree, and what search strategies are used by the state-of-the-art auto-scheduler for this domain.

Tensor Program Scheduling in Halide

Halide is a domain specific language for high-level specifying computer vision pipelines (e.g., Gaussian blur) as well as a tool that can automatically find high-performance low-level hardware-dependent implementations of the pipeline during compilation. It represents the pipelines as directed acyclic graphs (DAG), where each graph node is called a *stage*. A pipeline usually consists of multiple stages of computation; at each stage, two successive decisions need to be made respectively for *intra-stage ordering* and *cross-stage granularity* (Li et al. 2018; Adams et al. 2019). Given a pipeline, a complete low-level implementation is produced upon the decisions at all stages in the corresponding DAG have been fully made, and the process of such decision-making is referred to as Halide scheduling.

Figure 4 illustrate the auto-scheduling process for a toy pipeline — it contains three functional stages, so there are 6 decision depths. In practice, a real-world pipeline usually contains tens of functional stages, and each decision node may contain hundreds of actions (e.g., 100^{20}), such that exhaustive search is infeasible (Ragan-Kelley 2014; Li et al. 2018). The state-of-the-art $X86$ CPU auto-scheduler finds schedules using a variants of beam search along with a *cost model* (Adams et al. 2019).

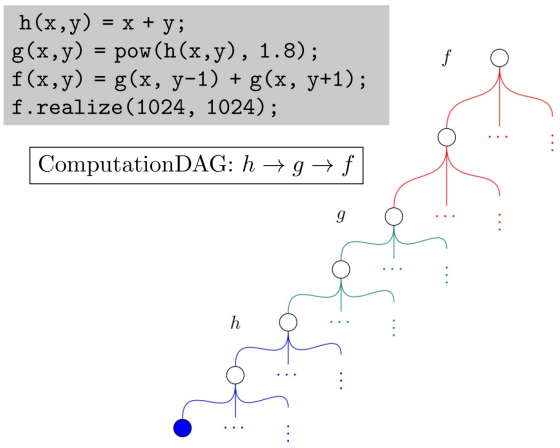


Figure 4: A toy Halide program and its scheduling space.

The major innovations of auto-scheduler (Adams et al. 2019) can be summarized as follows. First, it uses a pre-trained *cost model* to guide the search. The cost model hybridizes *symbolic analysis* and a neural network (LeCun, Bengio, and Hinton 2015) — for each node s in search tree, suppose $N(s)$ decisions have been made in s, then it calculates a vector of schedule features of size $N(s)$, then estimates a cost. It is generally true that those costs at depth $d + 1$ are greater than depth d. However, this monotonic increasing property is not truly guaranteed because of the use of neural network, implying that for heuristic search, the available heuristic is not *admissible*. These properties further imply that algorithms that utilize admissible pruning, e.g., IDA* (Korf 1985), are infeasible for the domain. Indeed, to enable large space search, an incomplete beam search is used in (Adams et al. 2019) [2]. More specifically, it develops a *multi-pass coarse-to-fine beam search*. That is, the auto-scheduler runs beam search multiple times, where each run uses a different hash function for *pruning* actions; thus, each pass defines a different *coarse region* for searching, and beam search pass $i + 1$ uses a hash set from pass i to further refine the region being searched. We note that, although local search algorithms, e.g., simulated annealing (Chen et al. 2018b) and genetic algorithms (Gao et al. 2021; Zheng et al. 2020), have also been used by other tensor compiler optimization frameworks (Chen et al. 2018a). These algorithms are not suited for the Halide compiler because currently it does not have direct and structured representation of a schedule object. The search space is a decision-tree, where one can only construct a schedule by sequentially making decisions. Therefore, *schedules* only occur at the last depth of the decision-tree. In other words, the schedules are just terminal states — no explicit structures are exposed for the

[2] Fast speed is demanded because auto-scheduling happens during program compilation.

search algorithms to define concepts such as neighboring schedules and schedule transformations.

MB2FBS for Halide

To have a fair comparison, we implement MB2FBS directly on the codebase of (Adams et al. 2019)[3]. We use the same pre-trained cost model for X86 CPU throughout our experimentation. In addition, since the cost model in Halide tends to give less comparable cost estimates for states in different depths, we define a new priority function that combines depth and cost.

For a given pipeline, the maximum search depth d_{max} is known prior to search, and suppose the smallest known solution cost is X. For state s, let $s.cost$ be the cost model estimate for s. To balance the influence of $s.depth$ and $s.cost$, we compute the priority of s as follows:

$$\rho(s) \triangleq \frac{X - s.cost}{D_{max} - s.depth},$$

To embed MB2FBS into the *multi-pass* search framework as (Adams et al. 2019), we set D and X as follows:

- We set $D_{max} \leftarrow i \cdot d_{max} + 1$, where i is pass index starting from 1.
- In first run $X \leftarrow \infty$ [4], in other runs, X set to the smallest known solution cost.

Figure 5 demonstrates how (D_{max}, X) modifies the priorities of states, resulting in a more favourable cross-depth comparison of states in Halide. By varying (D_{max}, X), MB2FBS is able to search different regions from pass to pass. Note that this priority function has no effect on beam search, because it only compares states in the same depth.

As recommended in Halide, the maximum pass number is set to 5 for both MB2FBS and (Adams et al. 2019). For beam search, the recommended beam size is 32, but we also include results from beam size 64. For MB2FBS, we test two variants: the β-controlled variant with $\beta_1 = 28, \beta_2 = 4, \beta = 32$, and uncontrolled variant $\beta_1 = 30, \beta_2 = 2$. We set memory bound to ∞ for all scenarios since we have seen this is safe choice for both controlled and uncontrolled MB2FBS.

Table 2 contains the results of multi-pass beam search and MB2FBS on 15 standard test pipelines in Halide. Among the 15 test cases, the uncontrolled version of MB2FBS is able to find 9 better costs than the best of beam-32 and beam-64, while having 3 cases with slightly larger costs. The results of the controlled version of MB2FBS are similar. Both versions achieved an overall geometric mean smaller than the beam search counterparts. We note that the costs in Table 2 are best terminal costs from cost model, but those costs are typically consistent with real schedule qualities — when translating the terminal states into Halide schedules, after benchmarking, we find that two MB2FBS variants achieved averaged time-costs respectively $21587\mu s$ and $21561\mu s$, while the beam-search-32 achieved $27821\mu s$, i.e., a 22% speedup is observed. Note that this improvement is non-trivial in the sense once a pipeline is compiled, it might be used by millions of users everyday, cumulatively saving considerable time and energy.

We further list the node expansions of all algorithms in Table 3. The uncontrolled version used slightly more node expansion than beam search with beam size 32, while for the controlled version, the node expansion is strictly smaller or equal. These results in Table 3 certify that, for the MB2FBS algorithms, the reason that they achieved better performance is mainly due to their incorporation of best-first seeking strategies, not more computation clocks.

[3]https://github.com/halide/Halide

[4]In implementation, we use 10^9 to represent ∞

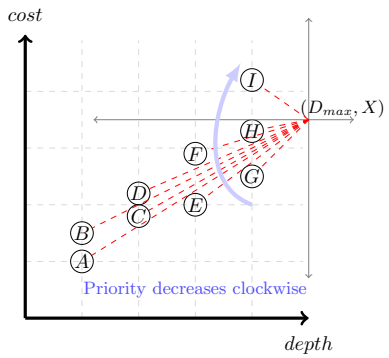


Figure 5: Illustration of the priority function that combines depth and cost estimate. Only according to cost, these nodes are prioritized by $A \succcurlyeq B \succcurlyeq C \succcurlyeq E \succcurlyeq D \succcurlyeq G \succcurlyeq F \succcurlyeq H \succcurlyeq I$ — with the new priority function, it becomes $G \succcurlyeq E \succcurlyeq A \succcurlyeq C \succcurlyeq B \succcurlyeq D \succcurlyeq F \succcurlyeq H \succcurlyeq I$.

Pipeline	Adams2019-b		MB2FBS-$(\beta_1, \beta_2, \beta)$	
	32	64	$(30, 2, \infty)$	$(28, 4, 32)$
bilateral grid	5.00327	5.00327	**4.85979**	*5.08237*
local laplacian	54.0256	55.7315	**45.3922**	**44.3335**
nl means	47.795	47.795	**46.3747**	**37.8478**
lens blur	10.4686	10.205	**10.1556**	**8.20918**
camera pipe	4.31618	4.02908	*4.39345*	*4.399*
stencil chain	77.213	77.2167	**75.7563**	**73.2237**
harris	1.73584	1.73584	*1.83047*	*1.73923*
hist	4.54293	6.70719	**4.0048**	**4.06191**
max filter	61.9724	58.9023	*61.9724*	*61.9724*
unsharp	4.43961	4.40831	**4.382**	**4.382**
interpolate	17.6231	17.5683	**17.2449**	**17.3213**
conv layer	23.36	23.36	23.36	23.36
iir blur	9.24476	9.54103	9.24476	9.24476
bgu	16.5894	16.5894	**16.5362**	*16.6023*
mat mul	11.5766	11.5766	11.5766	11.5766
GeoMean	13.29263	13.5584	12.963	12.57535

Table 2: Minimum solution cost found by MB2FBS and Beam Search on 15 standard test cases. Boldface indicates the corresponding result is superior to those from the two beam search runs.

Effectiveness of Modified Priority Function

To verify the effectiveness of the new priority function, we run additional experiments with the new priority function

Pipeline	Adams2019-b		MB2FBS	
	32	64	$(30, 2, \infty)$	$(28, 4, 32)$
bilateral grid	514.0	998.0	570.4	502.2
local laplacian	6594.0	13176.0	6722.2	5903.6
nl means	834.0	1660.0	1066.8	786.4
lens blur	4802.0	9572.0	5113.2	4317.2
camera pipe	2370.0	4723.0	2627.0	2244.0
stencil chain	2242.0	4451.0	2959.0	2178.4
harris	770.0	1510.0	923.2	752.6
hist	578.0	1136.0	729.0	570.8
max filter	578.0	1148.0	635.0	573.2
unsharp	578.0	1148.0	639.2	570.4
interpolate	3266.0	6524.0	3365.2	2932.4
conv layer	258.0	514.0	286.6	282.8
mat mul	194.0	358.0	223.8	217.4
iir blur	258.0	508.0	357.2	274.0
bgu	1282.0	2556.0	1292.8	1174.4
Average	1674.5	3332.1	1834.0	1552.0
Avg.Time(s)	106	164	106	85

Table 3: Each node expansion number is averaged over the 5 passes being used for each algorithm. Results obtained on same Intel CPU server.

Pipeline	MB2FBS-$(\beta_1, \beta_2, \beta)$ sol. and expansion			
	$(30, 2, \infty)$	#exp	$(28, 4, 32)$	#exp
bila. grid	4.86	674.8	5.08	502.2
local lap.	45.29	2047.2	42.90	5903.6
nl means	37.11	7128.2	40.54	786.4
lens blur	9.01	1711.2	9.36	4317.2
cam. pipe	4.33	5113.8	4.32	2244
ste. chain	71.60	2952.2	71.88	2178.4
harris	1.78	2739.4	1.74	752.6
hist	4.05	883.8	4.07	570.8
max filter	61.97	691.2	61.97	573.2
unsharp	4.38	620	4.38	570.4
inter.	17.33	1218.8	17.59	2932.4
conv.	23.36	3053	23.36	282.8
iir blur	9.245	235.8	9.24	274
bgu	16.73	549.2	16.75	1174.4
mat mul	11.58	274.4	11.58	217.4
GeoMean	12.61	1267.23	12.70	940.89
Average	21.51	1992.87	21.65	1551.99

Table 4: Minimum solution cost found and node expansion by MB2FBS on 15 standard test cases without using the new priority function.

being unused, i.e., it only prioritizes states by cost estimates. Table 4 shows the results. Referring to the results in Tables 2, we see that for the uncontrolled version, the overall averaged node expansion increased from 1834.0 to 1992.87, while the geometric mean cost reduced from 12.963 to 12.61. For the controlled version, the averaged node expansion remained almost unchanged, while the geometric mean increased from 12.575 to 12.70. From these results, it is clear that the new priority function helped the controlled version of MB2FBS, while for the uncontrolled version, its practice merit was unnoticed. This is unsurprising because that the major change brought by the new priority function is that it drives the search to visit deeper states preferably — this is a desirable property for the *controlled* version as it can be viewed as a truncated and early-stopped modification of uncontrolled MB2FBS.

Analysis of the Effect of Mixing Pushing Forward and Onward

While the results from previous sections indicate MB2FBS is superior to beam search for solving real-world Halide problems, because of the large problem sizes, it is not easy to conduct extensive ablation studies to fully show what cases MB2FBS is more preferable and what cases not. To clearly show how mixing pushing forward and onward affect search performance, we develop a synthetic tree environment. This is for mimicking Halide problems but the instances can be randomly generated with parameterization. The transparency nature of this environment allows us to conduct experiments for quantifying key strengths of MB2FBS. We can sample instances of different characteristics and scale by appropriate parameterization — for these problems, optimal solutions can also be retrieved and used as a golden reference.

Specifically, we randomly sample *synthetic tree* as a t-ary tree environment for empirically studying different search strategies. Each tree instance is randomly generated such that each internal node has exactly b_{max} children, and each node s has a randomly sampled cost drawn from $Uniform[0, depth(s))$, where $depth(s)$ is the depth of s. The tree has maximum depth of d_{max}. To increase solution difficulty, the cost for each leaf node l be sampled from $Uniform[depth(l) + \delta, depth(l) + \delta^2)$, where δ is a parameter. The path cost for any leaf node l is defined as the cumulative cost of all nodes along the path from *start* to l.

Typically, smaller δ value results random instances easier to solve for greedy-biased strategies. Figure 6 illustrates how δ affects search performances — when $\delta = 0$, pure greedy algorithm can easily find near optimal or optimal solution cost. Thus, the performances of all algorithms become less distinguishable, and the incorporation of *pushing onward* does not exhibit visible benefit. However, as δ increases, the advantage of MB2FBS becomes noticeable — for $\delta = 100$, MB2FBS obtains clearly better accuracy than the other algorithms including beam search with beam size of 32. To ensure the computation used by MB2FBS is comparable to beam search, we set $\beta_1 \leftarrow 30, \beta_2 \leftarrow 2$. We use maximum tree-depth $d_{max} = 8$ and branching factor $b_{max} = 4$ because we find this setting produces neither too large nor too trivial instances for experimentation and analysis.

Fixing $\delta = 100$, we then experimentally compare MB2FBS and beam search from various configurations. Figure 7 show the comparison of MB2FBS and beam search perform with varied beam size $b = \{8, 16, 32, 64, 128, 256\}$. In terms of accuracy, both MB2FBS variants (b-controlled and uncontrolled) perform better than beam search overall. For all algorithms, the node expansion grows linearly w.r.t b. For the uncontrolled MB2FBS $\beta_1 = b - 2, \beta_2 = 2$, the node expansion of MB2FBS is slighter larger than beam search. For the controlled version, MB2FBS never expands

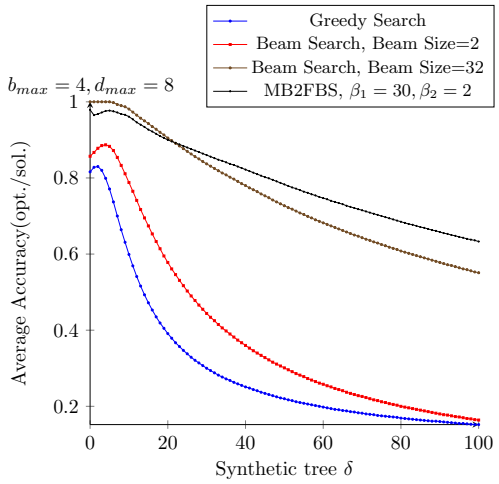


Figure 6: Solution accuracy decreases as δ increases. For each experiment δ, 10 random trees are sampled and then ran by each algorithm.

MB2FBS	Uncontrolled		256-controlled	
(β_2, β_1)	Accuracy	#exp	Accuracy	#exp
$0, 256$	0.939	1109.4	0.939	1109.0
$2, 254$	0.939	1114.6	0.939	1105.8
$8, 248$	0.945	1137.0	0.939	1098.0
$16, 250$	0.945	1173.0	0.939	1087.7
$32, 224$	0.952	1328.6	0.945	1073.8
$64, 192$	0.930	1462.6	0.714	1039.5
$128, 128$	0.942	1639.8	0.708	1024.4

Table 5: Fixing $\beta_1 + \beta_2 = 256$, results of MB2FBS with varying β_2. As a reference, the accuracy and node expansion for beam search with beam size of 256 are 0.939 and 1109.0 (equivalent to MB2FBS with $\beta_2 = 0, \beta_1 = 256$). All results are averaged over 10 independently random sampled synthetic trees.

more nodes than beam search on all cases. The better results of MB2FBS are due to the addition of beam search with a small amount of *pushing onward* behaviour. When beam size becomes large, i.e., $b = 256$, all algorithms achieve the same accuracy of 0.939, presumably because $\beta_2 = 2$ is insufficient to bring an effect to MB2FBS. To verify this, we run additional experiments by varying $\beta_2 = \{2, 8, 16, 32, 64, 128\}$. Table 5 shows the results — e.g., if we increase $\beta_2 = 32$, both controlled and uncontrolled MB2FBS are able to achieve an averaged accuracy better than beam search (0.952 and 0.945, respectively), meanwhile the node expansion of controlled MB2FBS (1073.8) is less than beam search.

Conclusions

We have described a memory-bounded best-first search beam algorithm framework where different search behaviours can be generated by correspondingly setting its

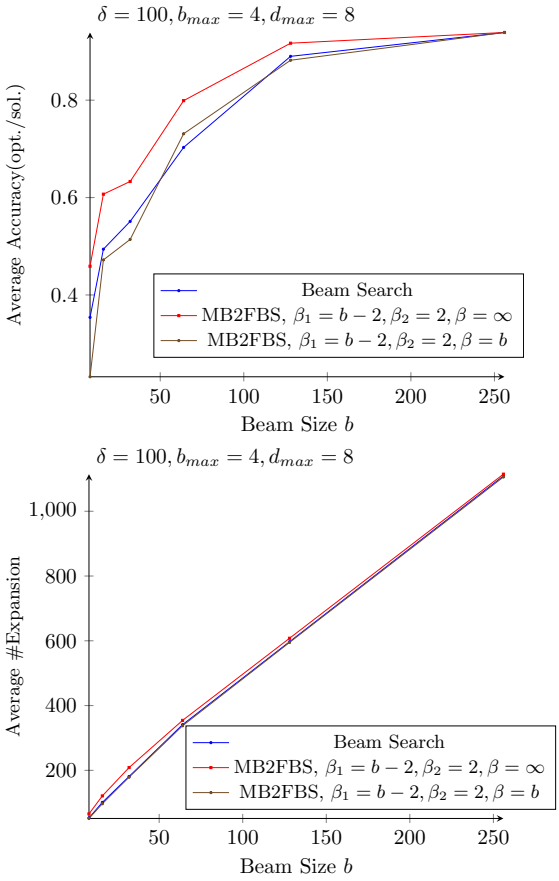


Figure 7: For beam search, the x-axis b represents beam size. For MB2FBS, b means $\beta_1 + \beta_2$, and a constant value of 2 is given for β_2. 10 random trees are generated and tried for each experiment.

hyper-parameters. We have provided an analysis on its properties and demonstrated the usefulness using artificially created tree problems. Most importantly, the advantage of the new algorithm is manifested in its successful deployment to auto-scheduling Halide programs; MB2FBS yielded better solutions without incurring extra computation budgets. While it is indisputable that no search strategies would be universally superior to others in all cases, a practically desirable solution might most likely be the one that is able to strike a balance between the extremes. From this perspective, the extra parameters introduced in MB2FBS should not be viewed a deficiency, rather, MB2FBS provides a new opportunity for practitioner to swiftly configure these parameters for problem instances of different characteristics.

Machine learning compilers have drawn a considerable attention recently (Li et al. 2020). The auto-scheduling algorithms typically combine search and cost model learning, e.g. (Chen et al. 2018b). Our results in Halide certifies that advanced heuristic search can be an valuable part for compiler optimization — we expect that more AI planning algorithms will be applied for these lines of development.

References

Adams, A.; Ma, K.; Anderson, L.; Baghdadi, R.; Li, T.-M.; Gharbi, M.; Steiner, B.; Johnson, S.; Fatahalian, K.; Durand, F.; et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4): 1–12.

Atkinson, M. D.; Sack, J.-R.; Santoro, N.; and Strothotte, T. 1986. Min-max heaps and generalized priority queues. *Communications of the ACM*, 29(10): 996–1000.

Birgin, E. G.; Ferreira, J. E.; and Ronconi, D. P. 2015. List scheduling and beam search methods for the flexible job shop scheduling problem with sequencing flexibility. *European Journal of Operational Research*, 247(2): 421–440.

Bisiani, R. 1992. Beam search. *Encyclopedia of Artficial Intelligence*, 1467–1468.

Chakrabarti, P. P.; Ghose, S.; Acharya, A.; and de Sarkar, S. C. 1989. Heuristic search in restricted memory (research note). *Artificial Intelligence*, 41(2): 197–222.

Chen, T.; Moreau, T.; Jiang, Z.; Zheng, L.; Yan, E.; Shen, H.; Cowan, M.; Wang, L.; Hu, Y.; Ceze, L.; et al. 2018a. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 578–594.

Chen, T.; Zheng, L.; Yan, E.; Jiang, Z.; Moreau, T.; Ceze, L.; Guestrin, C.; and Krishnamurthy, A. 2018b. Learning to Optimize Tensor Programs. In Bengio, S.; Wallach, H.; Larochelle, H.; Grauman, K.; Cesa-Bianchi, N.; and Garnett, R., eds., *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Felner, A.; Kraus, S.; and Korf, R. E. 2003. KBFS: K-best-first search. *Annals of Mathematics and Artificial Intelligence*, 39(1): 19–39.

Furcy, D.; and Koenig, S. 2005. Limited discrepancy beam search. In *IJCAI*.

Gao, C.; Mo, T.; Zowtuk, T.; Sajed, T.; Gong, L.; Chen, H.; Jui, S.; and Lu, W. 2021. Bansor: Improving Tensor Program Auto-Scheduling with Bandit Based Reinforcement Learning. In *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, 273–278. IEEE.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Harvey, W. D.; and Ginsberg, M. L. 1995. Limited discrepancy search. In *IJCAI (1)*, 607–615.

Kaindl, H.; and Khorsand, A. 1994. Memory-bounded bidirectional search. In *AAAI*, 1359–1364.

Karbowska-Chilinska, J.; Koszelew, J.; Ostrowski, K.; Kuczynski, P.; Kulbiej, E.; and Wolejsza, P. 2019. Beam search Algorithm for ship anti-collision trajectory planning. *Sensors*, 19(24): 5338.

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

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

Korf, R. E. 1996. Improved limited discrepancy search. In *AAAI/IAAI, Vol. 1*, 286–291.

LeCun, Y.; Bengio, Y.; and Hinton, G. 2015. Deep learning. *nature*, 521(7553): 436–444.

Li, M.; Liu, Y.; Liu, X.; Sun, Q.; You, X.; Yang, H.; Luan, Z.; Gan, L.; Yang, G.; and Qian, D. 2020. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3): 708–727.

Li, T.-M.; Gharbi, M.; Adams, A.; Durand, F.; and Ragan-Kelley, J. 2018. Differentiable programming for image processing and deep learning in Halide. *ACM Transactions on Graphics (TOG)*, 37(4): 1–13.

Medress, M. F.; Cooper, F. S.; Forgie, J. W.; Green, C.; Klatt, D. H.; O'Malley, M. H.; Neuburg, E. P.; Newell, A.; Reddy, D.; Ritea, B.; et al. 1977. Speech understanding systems: Report of a steering committee. *Artificial Intelligence*, 9(3): 307–316.

Meister, C.; Vieira, T.; and Cotterell, R. 2020. Best-first beam search. *Transactions of the Association for Computational Linguistics*, 8: 795–809.

Pearl, J. 1984. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Pub. Co., Inc., Reading, MA.

Ragan-Kelley, J.; Barnes, C.; Adams, A.; Paris, S.; Durand, F.; and Amarasinghe, S. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6): 519–530.

Ragan-Kelley, J. M. 2014. *Decoupling algorithms from the organization of computation for high performance image processing*. Ph.D. thesis, Massachusetts Institute of Technology.

Roy, A.; and Todorovic, S. 2014. Scene Labeling Using Beam Search Under Mutex Constraints. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Russell, S. 1992. Efficient memory-bounded search methods. In *ECAI '92 Proceedings of the 10th European conference on Artificial intelligence*, 1–5.

Zhang, W. 1998. Complete anytime beam search. In *AAAI/IAAI*, 425–430.

Zheng, L.; Jia, C.; Sun, M.; Wu, Z.; Yu, C. H.; Haj-Ali, A.; Wang, Y.; Yang, J.; Zhuo, D.; Sen, K.; et al. 2020. Ansor: Generating {High-Performance} Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 863–879.

Zhou, R.; and Hansen, E. A. 2002. Memory-Bounded A* Graph Search. In *FLAIRS conference*, 203–209.

Zhou, R.; and Hansen, E. A. 2005. Beam-Stack Search: Integrating Backtracking with Beam Search. In *ICAPS*, 90–98.