IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 3 December 2023, accepted 14 December 2023, date of publication 25 December 2023, date of current version 5 January 2024.

Digital Object Identifier 10.1109/ACCESS.2023.3346898

RESEARCH ARTICLE

K-Focal Search for Slow Learned Heuristics

MATIAS GRECO[1,2], JORGE TORO[2], CARLOS HERNÁNDEZ[1,3], AND JORGE A. BAIER[2,4]

[1]Facultad de Ingeniería, Arquitectura y Diseño, Universidad San Sebastián, Santiago 8420524, Chile
[2]Departamento de Ciencia de la Computación, Pontificia Universidad Católica de Chile, Santiago 7820436, Chile
[3]Centro Científico y Tecnológico de Excelencia Ciencia y Vida, Santiago 8580000, Chile
[4]Instituto Milenio Fundamentos de los Datos, Santiago 7820436, Chile

Corresponding author: Matias Greco (matias.greco@uss.cl)

This work was supported in part by Vicerrectoría de Investigación y Doctorados of Universidad San Sebastián, under Grant VRID_APC23/27; in part by the National Agency for Research and Development (ANID), Doctorado Nacional, under Grant 2019-21192036; in part by Centro Nacional de Inteligencia Artificial (CENIA), FB210017, BASAL, ANID; and in part by Centro Ciencia y Vida, FB210008, Financiamiento Basal para Centros Científicos y Tecnológicos de Excelencia, ANID.

ABSTRACT Bounded suboptimal heuristic search is a family of search algorithms capable of solving hard combinatorial problems, returning suboptimal solutions within a given bound. Recent machine learning approaches have been shown to learn accurate heuristic functions. Learned heuristics, however, are slow to compute; concretely, given a single search state s and a learned heuristic h, evaluating $h(s)$ is typically very slow relative to expansion time, since state-of-the-art learned heuristics are implemented as neural networks. However, by using a Graphics Processing Unit (GPU), it is possible to compute heuristics using batched computation. Existing approaches to batched heuristic computation are specific to satisficing search and have not studied the problem in the context of bounded-suboptimal search. In this paper, we present K-Focal Search, a bounded suboptimal search algorithm that in each iteration expands K states from the `FOCAL` list and computes the learned heuristic values of the successors using a GPU. We experiment over the 24-puzzle and Rubik's Cube using DeepCubeA, a very effective and inadmissible learned heuristic. Our results show that K-Focal Search benefits both from batched computation and from the diversity in the search introduced by its expansion strategy. Over standard Focal Search, K-Focal Search improves runtime by a factor of 6, expansions by up to three orders of magnitude, and finds better quality solutions, keeping the theoretical guarantees of Focal Search.

INDEX TERMS Bounded-suboptimal search, heuristic search, learned heuristics.

I. INTRODUCTION

There exists a variety of problems in AI that require the use of algorithms developed by the heuristic search community; e.g. A* [1] or an A*-like algorithm [2], [3]. Heuristic search relies on the use of a heuristic to guide search, which is a function h is such that $h(s)$ is a *cost-to-go* estimate, that is, it estimates the cost of path that reaches a goal state from s. In hard search problems, i.e. problems with a large state space, the heuristic function is invoked many times, and a substantial fraction of the time spent during the search is used to compute the heuristic. Therefore, computing h quickly is key to efficiency.

The associate editor coordinating the review of this manuscript and approving it for publication was Alberto Cano.

In the last few years, a number of machine learning techniques have been proposed to learn heuristic functions, which allows to find suboptimal results faster [4], [5], [6]. As the main technique used are neural nets, the evaluation of $h(s)$ can become quite slow. For example, in the 15-puzzle, the Manhattan Heuristic, which is a formula computable in linear time in the size of the puzzle size, takes on average 3 orders of magnitude less time that DeepCubeA heuristic, which is a very effective neural network heuristic trained with Reinforcement Learning. Indeed, our data shows that up to 80% of search runtime can be consumed computing a neural-net heuristic.

Slow heuristic computation has been tackled by previous work by exploiting the computational power of Graphics Processing Units (GPUs), which allows evaluating neural

© 2023 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

nets in parallel. Batched Weighted A* (BWAS) [6] is a search algorithm similar to Weighted A* but in which K states are extracted from the OPEN list in each search iteration.

Because BWAS [6] is based on A*, it does not provide optimality guarantees when used with black-box learned heuristics, which are typically inadmissible. Another related approach is BatchA* [7], which is a version of A* that uses batched heuristic computation. While the authors of BatchA* propose a method to learn admissible heuristics, their approach cannot exploit any given black-box learned heuristic without additional training.

In this paper we address the problem of exploiting slow learned heuristics on a branch of heuristic search known as *bounded suboptimal heuristic search*, which returns a solution whose suboptimality is within a given bound. We propose K-Focal Search (K-FS) a new search algorithm which generalizes Focal Search [8] addressing the efficiency issues raised by the use of black-box inadmissible learned heuristics by using a GPU while providing guarantees on solution quality. In addition, by increasing exploration, compared to FS, it may reduce expansions substantially, avoiding regions of the state space where the learned heuristic is not well informed. Instead of expanding the best state in `FOCAL`, it expands the best K states in `FOCAL`, producing a batch of successors, whose heuristic values are computed by the GPU. After extraction, if the goal has not been found, all K states are sequentially expanded, and the heuristic values for the union of their successors are evaluated via batched computation. Batched computation reduces heuristic computation times per state significantly, ultimately reducing total runtime.

We prove that K-FS is complete and w-optimal and study additional theoretical properties of the algorithm. Empirically, we show the applicability of K-FS solving two standard benchmarks: the 24-puzzle and the Rubik's cube, using DeepCubeA [6], a very effective learned heuristic.

Our empirical results show a reduction of up to three orders of magnitude in the number in of expansions compared to FS. Such large effects on runtime had not been observed in the past with algorithms that expand multiple states from the Open list, like KBFS [9] in combination with standard heuristics. Also, K-FS can reduce the time spent to compute the learned heuristic up to three orders of magnitude, and return solutions 10 times faster.

K-FS is related to other algorithms that exploit parallelism within search [10], [11], [12], [13]. Perhaps the approach most related is PA*SE [14] which is designed for search problems with slow *expansions*, and thus parallelizes expansions, which still sequentially evaluates the heuristic values of successor states.

The main contributions of this paper are:

1) We propose K-FS, a new bounded suboptimal algorithm which exploits batched GPU computation to address the problem of slow heuristic computation. The most related algorithms in the literature to K-FS are those proposed by Spies et al. [15] and Araneda et al. [16], which use of neural-net inadmissible heuristics in combination with admissible heuristics in a Focal Search (FS) framework. However, these approaches do not address the problem of slow heuristic computation.
2) We analyze the theoretical properties of K-FS.
3) We evaluate the proposed algorithm in two standard benchmarks for different suboptimality bounds and compare it with other bounded suboptimality search algorithms.

This article extends a previous SoCS-22 extended abstract [17]. The following items describe materials not included in the previous publication.

- A complete description of the algorithm, which include the pseudocode.
- An analysis of the theoretical properties of the algorithm.
- An extended experimental analysis, which includes two domains: 24-Puzzle and the Rubik's Cube, with two differents suboptimality bounds.

The rest of the paper is organized as follows. We start by describing the most essential background concepts, which include search and neural network heuristics. Then, we present the new algorithm. We continue with the empirical analysis and finalize with conclusions.

II. BACKGROUND

In this section we review the backround for the rest of the paper.

A. SEARCH

A *search task* is a tuple $P = (G, s_{start}, s_{goal})$, where G is a search graph, $s_{start} \in S$ is the *start state*, and $s_{goal} \in S$ is the *goal state*. A *search graph* has the form $G = (S, E, c)$ where S is a set of states, $E \subseteq S \times S$ is a finite set of edges, and $c : E \to \mathbb{R}^{\geq 0}$ is a cost function associating a non-negative cost with every edge.

A path is a sequence of states $\sigma = s_1 s_2 \ldots s_n$, whose cost is $c(\pi) = \sum_{i=1}^{n-1} c(s_i, s_{i+1})$. We denote the cost of a minimum-cost path from s to s_{goal} by $h^*(s)$. A solution (resp. optimal solution) to a search task is a path (resp. minimum-cost path) connecting s_{start} and s_{goal}. A solution σ is *w-optimal*, where $w \geq 1$, when its cost does not exceed the cost of an optimal solution multiplied by w.

A heuristic h is a function $h : S \to [0, \infty)$ such that $h(s)$ estimates the cost of a path from s to s_{goal}. A heuristic h is *admissible* iff $h(s) \leq h^*(s)$, for every $s \in S$.

B. NEURAL-NET HEURISTICS

A *neural network* (NN) is a function which map a set of input values to a output values, such that $g = \sigma(Wx + b)$, where x is the input vector, W the matrix of trainable weights, b the bias offset vector, g the hidden layer, and $\sigma(.)$ a non-linear activation function. The output of the neural network is

TABLE 1. Time spending for calculate the heuristic estimates for a number of Rubiks cube states in CPU, GPU on sequential mode and GPU per batches, using DeepCubeA in a tesla K80 GPU from Google Colab.

Batch Size	CPU time/states	GPU seq time/states	GPU batch time/states	(batch - seq) time saved
1	2.74E-03	3.81E-03	3.64E-03	1.05E+00
10	3.00E-03	3.80E-03	4.54E-04	8.37E+00
100	2.45E-03	3.27E-03	7.38E-05	4.43E+01
1,000	2.51E-03	2.93E-03	2.95E-05	9.93E+01
10,000	2.44E-03	2.78E-03	2.33E-05	1.19E+02
100,000	2.41E-03	2.76E-03	3.72E-05	7.41E+01

obtained by $\hat{y} = Vg + b$, with a weight matrix V and output bias vector b [18]. Every layer before the output is called hidden layer. A deep neural network is composed by adding many hidden layers, where the output of the previous layer is used as input for the next layer. The computational complexity of a deep neural network is governed by matrix-vector multiplications.

A *NN heuristic h* is a heuristic computed by a neural network. During training, the weights W are updated to minimize a loss function that represents the difference between $h(s)$ and $h^*(s)$. Usually, NN heuristics are implemented as deep neural networks and trained with stochastic gradient descent and reinforcement learning [6], [19] or imitation and supervised learning [20], [21]. Learned heuristics are inadmissible; this is due to the inductive nature of the deep NNs.

A *Graphics Processing Unit (GPU)* is a piece of hardware containing thousands of processing units (cores). GPUs can parallelize matrix multiplications and therefore can evaluate the output of a number of neural nets in parallel. To take advantage of the power of a GPU, it is necessary to provide a batch of inputs whose neural network outputs can be calculated in parallel, which in our case are search states.

Table 1 shows the time spent to compute the heuristic values per each state using different batch sizes and different modes (i.e., the hardware where it is calculated). Using a GPU per batch, we observe that the time spent dramatically decreases up to two orders of magnitude as batch size increases, while the time used sequential methods remains constant. When the batch size exceeds the memory available per gpu, the time per state does not improve because it is converted to mini-batches.

C. KBFS AND BWAS

K-Best-First Search (KBFS) [9] is a generalization of Best-First Search (BFS). KBFS expands the best k states from the OPEN list in each expansion cycle. The difference with BFS is that, in BFS, after the successors of the best state are added to OPEN, a new cycle begins. In KBFS, the successors of a state are not examined until the rest of the previous K best states are expanded, and their successors are added to OPEN. We refer to KBFS with $K = k$ as KBFS(k).

The value of k has influence on the properties of the resulting algorithm. KBFS(1) is BFS, and KBFS(∞) is a Breadth-First search. Felner et al. [9] claim that KBFS can avoid the poor decisions made by BFS in zones of the state spaces where the heuristic has large errors.

Batched Weighted A (BWAS)* was proposed with the DeepCubeA learned heuristic [6]. BWAS is a specification of KBFS where, in each iteration, a batch of the best K states is expanded from OPEN. This algorithm uses the WA* evaluation function $f(s) = g(s) + w * h(s)$ to choose the best states to expand. It exploits GPU batch processing to compute the learned heuristic, which is implemented as a neural network, calculating the h values of all generated states in each expansion cycle in parallel. Both algorithms are closely related to K-FS, but none of these algorithms provides suboptimality guarantees if a NN heuristic is used, even in their most basic form with $K = 1$, because the learned heuristic is not admissible.

The structure of BWAS algorithm is similar to WA*. The algorithm extracts the best K states in OPEN and adds them to an auxiliary set to generate their successors. If the learned heuristic of the generated states has not been calculated, it is added to a batch to compute their heuristic in a GPU and inserted to OPEN with $f = g + wh$ as its evaluation function.

D. FOCAL SEARCH

Focal Search (FS) [8] is bounded-suboptimal heuristic search algorithm that, in each iteration, expands a single state from the FOCAL list until goal state is selected for expansion. As WA*, FS receives a parameter w to control the suboptimality of the returned solution. FS maintains two priority queues: OPEN and FOCAL. FOCAL contains the subset of states of OPEN which satisfy a sub-optimality bound w, given the information currently gathered by search; specifically, it contains all states $s \in$ OPEN such that $f(s) \leq w * f(top)$, where *top* is the element at the top of *Open*. Similar to A*, the OPEN list is sorted by $f(s) = g(s) + h(s)$ where $h(s)$ is an admissible heuristic function. FOCAL list is sorted by $h_{\texttt{FOCAL}}$, an arbitrary priority function that can be inadmissible. FS can acquire different behaviors according to how the suboptimality bound w is configured. With $w = \infty$, FS behaves like Greedy-BFS guiding the search by $h_{\texttt{FOCAL}}$. On the other hand, with $w = 1.0$ FS acts like A* breaking ties by $h_{\texttt{FOCAL}}$.

III. K-FOCAL SEARCH

K-Focal Search (K-FS(k)) is a generalization of Focal Search. Instead of selecting one state for expansion, K-FS(k) extracts the best k states from FOCAL, and unless the goal is among the extracted states, it expands all such states, computing the heuristic values of all their successors using GPU batched computation. In case where the FOCAL list contains less than k states, it selects for expansion all states in the FOCAL list. K-Focal Search with $k = 1$ [K-FS(1)] executes the exact procedure that Focal Search, but computes the heuristic values of all the successors of the expanded state in parallel using the GPU.

Algorithm 1 K-Focal Search

```
Input: A search task P = (G, s_start, s_goal),
       an admissible heuristic h, a suboptimality
       bound w, a neural-net heuristic h_FOCAL, a K
       parameter
Output: A goal node reachable from n_start
1  foreach s ∈ S do
2      g(s) ← ∞
3  g(s_start) ← 0
4  parent(s_start) ← null
5  f(s_start) ← h(s_start)
6  CLOSED ← ∅
7  Insert s_start to OPEN and FOCAL
8  while FOCAL is not empty do
9      f_min ← f-value of node at the top of OPEN
10     K' = max(K, | FOCAL |)
11     TO-EXPAND ← empty set
12     while |TO-EXPAND| < K' & FOCAL ≠ ∅ do
13         Extract s from FOCAL which minimizes
           h_FOCAL
14         Remove s from OPEN
15         Add s to CLOSED
16         if s = s_goal then
17             return s
18         Add s to TO-EXPAND
19     BATCH ← empty set
20     foreach s ∈ TO-EXPAND do
21         foreach t ∈ Succ(s) do
22             if g(s) + c(s, t) < g(t) then
23                 parent(t) ← s
24                 g(t) ← g(s) + c(s, t)
25                 f(t) ← g(t) + h(t)
26                 Insert t into OPEN
27                 if f(t) ≤ w * f_min then
28                     Add t to BATCH
29     top ← state at the top of OPEN
30     if f_min < f(top) then
31         foreach n ∈ OPEN do
32             if w * f_min < f(n) ≤ w * f(top) then
33                 Add n to BATCH
34     Compute the h_FOCAL-values of states in BATCH
       on GPU
35     Insert each state in BATCH to FOCAL
36 return "no-solution"
```

The suboptimality bound (w) configured in K-FS plays an important role in the algorithm. K-FS with larger values of w can perform more batched computation, because more states enter FOCAL. K-FS with $w = 1$ essentially performs an A* search, breaking ties by the h_{FOCAL}, and computes the learned heuristics values on the GPU.

Algorithm 1 shows the pseudocode of K-FS. The algorithm receives two heuristics as input: an admissible heuristic h, and a neural-net heuristic h_{FOCAL}. In Line 10, the algorithm chooses a K' value as the maximum between the configured K and the size of FOCAL. In Lines 12–17, the K best states in FOCAL are selected for expansion. If a chosen node for expansion is a goal state, then it is returned, otherwise, it is added to TO-EXPAND list.

Lines 19–28 correspond to the generation of successor states. For each state s in the TO-EXPAND list, the algorithm expands s, adding the successors to OPEN. If the successor $t \in Succ(s)$ satisfies the suboptimality bound (i.e. it is such that $f(t) \leq w * f_{min}$), then t is added to BATCH. Since the value of f_{min} may increase during execution and states that are in OPEN are not in FOCAL, Lines 29–33 verify if f_{min} increased and insert in BATCH the states that must be inserted in FOCAL because are within the suboptimality bound. Finally, the h_{FOCAL}-values of all states in BATCH are computed using the GPU and added to the FOCAL list.

Besides the computation of h_{FOCAL} using batched computation, an important difference between K-FS and FS is that more states are expanded in the same *expansion cycle*; that is in the same iteration of the while loop of Line 8. This has the potential of radically changing the order in which states are expanded, since the children of states that would have not been expanded by FS are added to FOCAL.

In summary, K-FS introduces two significant changes to FS. First, it increases the exploration power of FS by expanding the best k states from FOCAL; without losing theoretical properties, as we see below. Second, it exploits the batch processing capabilities of a GPU, reducing the time required. As we see later in our experimental evaluation, the first change implies a reduction in the number of expanded states, because, unlike FS, KFS is able to explore areas of the search space that are not necessarily explored by FS. When the learned heuristic is not well informed this results in finding solutions much earlier. The second change has a significant impact in the runtime of KFS.

Other algorithms such as BWAS and KBFS perform a similar method to expand various states in the same iteration. Nevertheless, they cannot provide suboptimality guarantees, even with an admissible heuristic, because they cannot ensure that the selected states are the best in the search frontier or are within the bound. However, K-FS uses f_{min} to prove the suboptimality of each state, and every state that is in FOCAL are within the bound, providing suboptimality guarantees.

IV. PROPERTIES OF K-FS

K-Focal Search has the following theoretical properties, which are significantly related to the properties of Focal Search and KBFS. First, K-FS, just like FS, is w-optimal.

Theorem 1: K-FS is complete and w-optimal using an admissible heuristic to sort OPEN.

Proof: Because K-FS does not change the conditions under which a state is added to FOCAL or OPEN, every state s in FOCAL is such that $f(s) < w \min_{t \in Open} g(t) + h(t)$ and

thus when a goal is extracted from FOCAL, it follows that the suboptimality bound is met. □

This following proposition could be helpful to estimate of how many states will be generated at each expansion cycle and the memory needed.

Proposition 1: During successive expansion cycles in which f_{min} remains constant, K-FS(∞), expands states in FOCAL in breadth-first order. As a consequence, when f_{min} remains constant, then every path in FOCAL at expansion cycle $i+1$ is a path that was in FOCAL at expansion cycle i, extended with an additional edge.

Proof: At each expansion cycle, K-FS(∞) expands all states in FOCAL. In the next expansion cycle and before f_{min} changes, all states in FOCAL will be the successors of states generated in the previous iteration. Once f_{min} has changed, the states in FOCAL may have been generated in any previous expansion cycle. □

Proposition 1 is related to a property of KBFS(∞), which establishes that KBFS(∞) is equivalent to Breadth-First search since, at each expansion cycle, it expands all states in the OPEN list, and all states in OPEN has the same depth. For K-FS(∞), because not all states in OPEN are in FOCAL, this property is maintained just until the f_{min} changes.

The follow theorem establishes that a specific configuration of K-FS does not perform more expansions cycles than regular FS. On problems with unitary cost, the minimum length of the solution is equal to the cost of the optimal solution. As we see in our experimental evaluation, it is usually the case that K-FS performs significantly fewer expansion cycles than FS.

Theorem 2: Let $EC(A)$ be the number of expansion cycles needed to solve a particular search instance by algorithm A. Then, given the same search instance and weight, the following relation holds:

$$EC(\text{K-FS}(\infty)) \leq EC(\text{FS}).$$

Proof: We start off by proving that the following three inequalities hold throughout the execution of the algorithm:

$$\text{FOCAL}^{FS} \subseteq \text{FOCAL}^{KFS} \cup \text{CLOSED}^{KFS}, \quad (1)$$
$$\text{OPEN}^{FS} \subseteq \text{OPEN}^{KFS} \cup \text{CLOSED}^{KFS}, \quad (2)$$
$$f_{min}^{FS} \leq f_{min}^{KFS}, \quad (3)$$

where FOCAL^{FS} and FOCAL^{KFS} denote the contents of the FOCAL list of FS and K-FS(∞), respectively, OPEN^{FS} and OPEN^{KFS} denote the contents of the OPEN list of FS and K-FS(∞), respectively, and CLOSED^{FS} and CLOSED^{KFS} denote the contents of the CLOSED list of FS and K-FS(∞), respectively.

Specifically we prove that at every *expansion cycle* of FS, (1), (2), and (3) hold true. We do this by induction on the number of expansion cycles. We assume that it an algorithm terminates at expansion cycle j, then for every expansion $k \geq j$, the data structures remain constant.

Our proof is by induction on the number of expansion cycles. For the base case (0 expansion cycles), both properties hold because all data structures and variables are identical in both algorithms.

Induction. We assume that we are at expansion cycle i and that the property has been held true in every expansion cycle, up to cycle i.

Now we prove that (1) and (2), hold at expansion cycle $i+1$. Specifically, we show that if FS expands s, then each successor t of s that is added to FOCAL^{FS} at the cycle i will be in $\text{FOCAL}^{KFS} \cup \text{CLOSED}^{KFS}$ at cycle $i+1$. Furthermore, we show that each successor t that is added to OPEN^{FS} at cycle i will be in $\text{OPEN}^{KFS} \cup \text{CLOSED}^{KFS}$ at cycle $i+1$. Let s be the state that is expanded by FS at cycle i. We identify two cases.

1) s is in FOCAL^{KFS}
2) s is in CLOSED^{KFS}

For case 1. If s is expanded by FS, s is also expanded by K-FS(∞). Now we prove that every successor of s that is added to FOCAL^{FS}, is added to FOCAL^{KFS}, and therefore is added to OPEN^{KFS} or is in CLOSED^{KFS}.

Let t be a successor of s. Because K-FS(∞) expands s, t may be added to FOCAL^{KFS} (it may not be added if t was generated by KFS in a previous cycle), which implies that conditions (1) and (2) hold true at expansion cycle $i+1$. If t has been generated in a previous expansion cycle, then t is either in FOCAL^{KFS} and OPEN^{KFS} or in CLOSED^{KFS} at cycle i (because $f_{min}^{FS} \leq f_{min}^{KFS}$). This implies that at cycle $i+1$, t belongs to CLOSED^{KFS}, which implies that the conditions (1) and (2) are satisfied at expansion cycle $i+1$.

For case 2, let t be a successor of s that is added to OPEN^{FS} at expansion cycle $i+1$. Since $s \in \text{CLOSED}^{KFS}$ at cycle i, then at some previous expansion cycle j ($j < i$), K-FS expanded s and added t to OPEN^{KFS}, unless t was already in CLOSED^{KFS}. This means that at every expansion cycle after j, state t is in $\text{OPEN}^{KFS} \cup \text{CLOSED}^{KFS}$. This implies that at cycle $i+1$ (2) is fulfilled. Finally, suppose that at cycle $i+1$ it holds that $t \in \text{FOCAL}^{FS}$ then $t \in \text{FOCAL}^{KFS}$ also holds, because $f_{min}^{FS} \leq f_{min}^{KFS}$. This implies that (1) holds true at cycle $i+1$.

To prove that (3) holds true at cycle $i+1$, we observe that if the property (2) holds true in $i+1$, then each state s that is in OPEN^{KFS} or is a state that is in OPEN^{FS} or is a descendant of a state t that belongs to OPEN^{FS}. Also, if s is in OPEN^{FS} but not in OPEN^{KFS}, then some (possibly empty) set of descendants of s are found in OPEN^{KFS}. On the other hand, since h is consistent, $f(s) \leq f(t)$ if t is a descendant of s. Therefore, at cycle $i+1$, $f_{min}^{FS} = \min_{s \in \text{OPEN}^{FS}} f(s) \leq \min_{s \in \text{OPEN}^{KFS}} f(s) = f_{min}^{KFS}$.

We have proven that (1), (2), and (3) hold true throughout execution. To finish the proof for the theorem, we observe that if FS terminates at a certain expansion cycle k, then the goal state is either in FOCAL^{KFS} or in CLOSED^{KFS}. If the former holds, KFS returns at expansion cycle k. If the latter holds, it returns the goal at an expansion cycle less than k, which finishes the proof. □

TABLE 2. Results on 24-puzzle.

	w=1.15							w=1.5						
	Cov.	EC	Exp	Cost	Time	h time	htis(%)	Cov.	EC	Exp.	Cost	Time	h time	htis(%)
WA*	0	-	-	-	-	-	-	68	-	1413332	112.10	602.88	-	-
DPS	0	-	-	-	-	-	-	80	-	901069	114.03	394.71	-	-
FS (seq)	0	722163	722163	-	1800	1.9E-03	79.84	96	4891.9	4891	110.23	13.89	2.3E-03	76.80
K-FS(1)	0	959647	959647	-	1800	1.3E-03	72.04	96	4891.9	4891	110.23	9.47	1.1E-03	61.55
K-FS(5)	100	852.5	4256	85.92	4.25	2.7E-04	33.68	100	105.1	519	102.68	0.81	2.0E-04	30.73
K-FS(10)	100	158.0	1561	85.22	1.57	1.4E-04	22.11	100	102.4	1004	100.90	1.00	9.1E-05	22.13
K-FS(25)	100	112.9	2751	84.38	2.02	5.2E-05	11.86	100	101.4	2463	100.26	2.01	3.8E-05	11.34
K-FS(100)	100	109.0	10467	83.44	7.00	1.6E-05	3.87	100	100.5	9618	99.26	5.96	1.1E-05	4.09
K-FS(120)	100	108.5	12468	83.32	8.12	1.4E-05	3.59	100	100.5	11516	99.19	7.71	1.1E-05	3.82
K-FS(240)	100	106.9	24392	82.94	14.97	9.0E-06	2.47	100	100.2	22765	98.74	14.84	9.0E-06	3.13

TABLE 3. Results on Rubik's cube.

	w=2.5							w=4.0						
	Cov.	EC	Exp	Cost	Time	h time	htis(%)	Cov.	EC	Exp.	Cost	Time	h time	htis(%)
WA*	0	-	-	-	-	-	-	0	-	-	-	-	-	-
DPS	0	-	-	-	-	-	-	1	-	266084*	29.00*	1789.00*	-	-
FS (seq)	9	59870	59870	28.0	755.22	2.0E-03	48.57	81	18815	18815	40.17	259.33	1.9E-03	56.49
K-FS(1)	11	59870	59870	28.0	501.69	4.3E-04	17.20	86	18815	18815	40.17	158.20	3.5E-04	18.45
K-FS(5)	25	15215	76073	27.0	569.36	1.1E-04	5.79	95	3765	18824	39.60	140.84	7.8E-05	5.48
K-FS(10)	31	5474	54736	27.0	398.42	6.0E-05	3.15	99	1290	12900	38.76	94.09	3.3E-05	2.94
K-FS(25)	50	438	10924	26.5	77.20	2.0E-05	1.69	100	130	3214	35.09	23.24	1.2E-05	1.39
K-FS(100)	75	56.8	5487	24.5	39.19	8.0E-06	0.89	100	45.0	4312	30.58	31.17	7.0E-06	0.99
K-FS(120)	77	62.8	7288	24.5	51.55	8.0E-06	0.80	100	37.9	4310	30.50	31.33	7.0E-06	0.97
K-FS(125)	76	61.0	7368	24.5	52.29	8.0E-06	0.79	100	38.5	4557	30.42	32.52	7.0E-06	0.94
K-FS(240)	85	38.0	8518	25.0	62.15	7.0E-06	0.72	100	30.4	6702	27.86	51.70	6.0E-06	0.82
K-FS(480)	89	42.3	18958	25.0	133.09	6.0E-06	0.68	100	25.2	10776	25.17	76.76	6.0E-06	0.84
K-FS(960)	95	24.0	20278	24.0	147.42	6.0E-06	0.73	100	24.9	20228	23.94	150.08	6.0E-06	0.78

V. EMPIRICAL RESULTS

Our empirical evaluation seeks to evaluate the performance of our algorithm and the impact of the k parameter with two different suboptimality bounds.

We evaluated our algorithm on two domains: the 24-puzzle and Rubik's cube. We use the publicly available trained models of DeepCubeA [6] as learned heuristic for both domains.

All algorithms were implemented in Python 3, and the experiments were run on an Intel Xeon E5-2630 machine with 64GB RAM, using a single CPU core and one GPU Nvidia Quadro RTX 5000. For all experiments, we use a 30-minute timeout.

Our algorithm was compared with FS in sequential mode (*FS*) using the same neural-net heuristic, which calculates the $h_{\texttt{FOCAL}}$ for each state that is inserted in `FOCAL` at the moment that it is inserted. Furthermore, we compare to two other state-of-the-art bounded suboptimality search algorithms: Weighted A* (*WA**) and Dynamic Potential Search (*DPS*) [22], which is a bounded-suboptimal version of potential search [23]. In our experimental evaluation, we do not include Explicit Estimation Search [24], which is another state-of-the-art bounded suboptimal search algorithm, because, as its authors assert, it does not significantly outperform WA* in domains with unitary costs, like the ones used in this evaluation. We also not include wGePA*-SE [25]. Even though this algorithm exploits parallel computation, it does not aim to parallelize the computation of the heuristic neither does it use a GPU for parallel computation.

K-FS was tested with different k values, from $k = 1$ to 960. We did not include K-FS(∞) (i.e., K-FS which expands all states in `FOCAL`) because it runs out of memory on some instances.

The result tables show the coverage (percentage of solved instances), average expansion cycles (EC), average expansions (exp.), average solution cost in the solved instances, average runtime, average time spent to compute the learned heuristic per state (h time), and the percentage of the search time which was used to compute the learned heuristic (htis [heuristic time in search]), for the instances solved by all algorithms in the domain, per each suboptimality bound. If an algorithm does not solve a particular instance with a suboptimality bound, it is not included in the table.

A. 24-PUZZLE

This is the 5 version of the classic sliding-tile puzzle. As admissible heuristics estimators, we use the Linear Conflict Heuristic [26]. For the evaluations, we use Korf's 50 instances for the 24-puzzle [27]. Table 2 shows the results in the 24-puzzle domain using two suboptimality bounds. With a small bound (i.e. $w = 1.15$), we observe that the

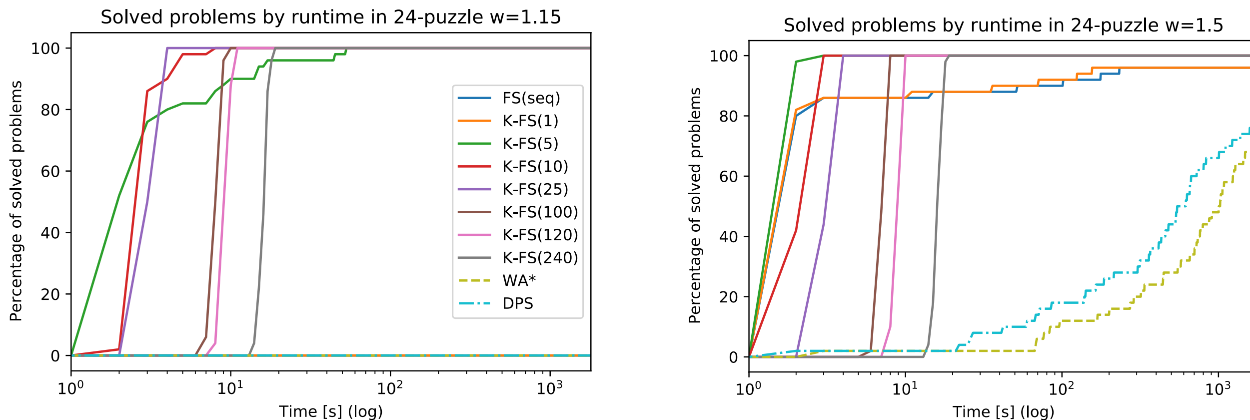

FIGURE 1. Results on 24-puzzle.

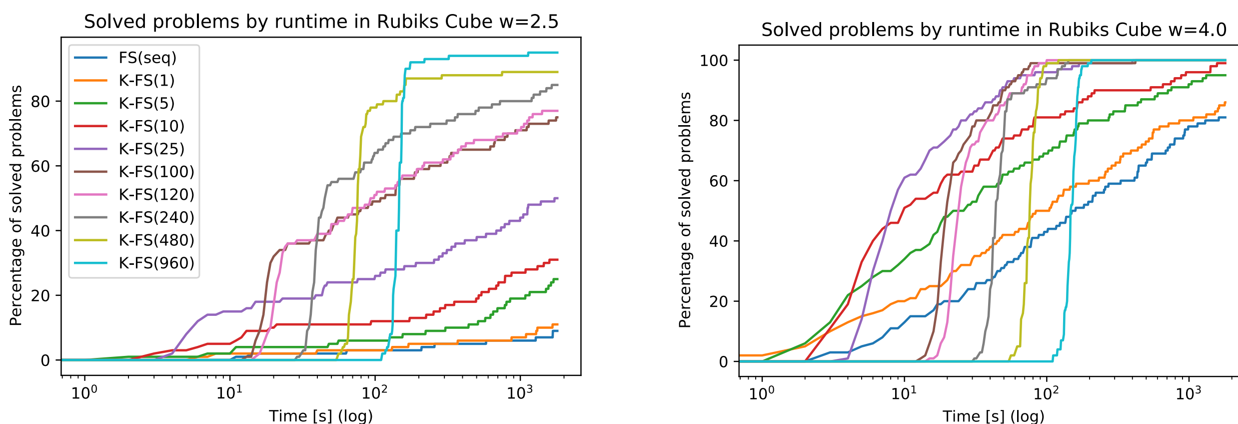

FIGURE 2. Results on Rubik's cube.

algorithms that do not use the learned heuristic (i.e., WA* and DPS) cannot solve instances. FS and K-FS(1), which use the learned heuristic, cannot solve instances. Both execute the same procedure, but due to the GPU parallelization, K-FS(1) can perform more expansions in the same runtime, resulting in a lower time spent to generate a state. Of the total time used in the search (1,800 seconds), in FS(seq), we observe that 79.7% was used only in calculating the heuristic, which decreases to 72% with K-FS(1). As k increases between 5 and 240, K-FS improves and can solve all problems in the set. K-FS(10) performs, on average, 3 orders of magnitude fewer expansions than K-FS(1), and finds solutions with better cost. In addition, K-FS reduced the percentage of time used to compute the heuristic from 72.04%, with $k = 1$, down to 2% with $k = 240$. An important observation is that, as k increases, the number of expansion cycles and the time spent calculating the heuristic for each state decrease, but the number of expansions and the runtime reaches its minimum in $k = 10$.

With suboptimality bound $w = 1.5$, WA* and DPS obtains just 68% and 80% of coverage, resp. FS (seq) and K-FS(1) 96% of coverage and find the same solutions, but K-FS can perform more expansions with the same runtime. As k increases, K-FS can obtain 100% of coverage, show better performance, finding better solutions with fewer expansions, and spend less time generating a state. In this case, K-FS(5) can perform one order of magnitude less expansions than K-FS(1) and three orders of magnitude than DPS and WA*. Same as the previous suboptimality bound, as higher K, the time spent computing the heuristic dramatically decreases, but the best results in terms of expansions and runtime are in $k = 5$.

Figure 1 shows the coverage vs. runtime per each algorithm in both suboptimality bounds. In the figure, with suboptimality bound $w = 1.15$, we observe that K-FS(25) solves the complete problem set in less than 5 seconds per problem. Similar behavior occurs in $w = 1.5$, where K-FS(5) and K-FS(10) solve all problems in less than 2 seconds per problem.

B. RUBIK'S CUBE

This is the classic $3 \times 3 \times 3$ combinatorial puzzle. As admissible heuristics estimators, we use a Pattern Database [28]. The pattern database, as the author recommends, was separated into three databases: one for the eight corner cubies (which has $8! * 3^7$ states) and two for the twelve edge cubies

(each has $12!/6! * 2^6$ states). The pattern databases were populated with Breadth-First Search. The final heuristic estimate is obtained by getting the maximum between the values stored in the three pattern databases for the state. For the evaluations, we use 100 random instances from the DeepCubeA test set, which was generated by randomly scrambling the goal state between 1,000 and 10,000 times. Table 3 shows the results on the Rubik's Cube domain using two different suboptimality bounds: $w = 2.5$ and $w = 4.0$. In this problem, we choose higher suboptimality bounds than in the 24-puzzle because the state space is larger and the admissible heuristic is less informed. With suboptimality bound $w = 2.5$, WA* and DPS cannot solve instances within the deadline. On the other hand, FS (seq) could solve 9% of the instances and K-FS(1) just two more instances because batch processing accelerates the computation of the learned heuristic. On average, between the instances solved by all algorithms, FS (seq) spent 48.57% of the search time calculating the learned heuristic, and K-FS(1) reduces the time to 17.20%.

As k increases, the coverage increases up to 95% using K-FS(960), which, on average, is one order of magnitude fewer expansions than K-FS(1), finding better cost solutions. The time spent computing the heuristic, per each state, was reduced by almost three orders of magnitude as k increases, and it went from 48% of the search time to only 0.7%. An important observation is that the number of expansion cycles performed by K-FS(960) is equal to the solution cost, which in practice means that K-FS(960) behaves similar to K-FS(∞), and according to Proposition 1, which expands the search tree in Breadth-First order. In the selected instances, with $w = 2.5$, the best configuration of K in terms of expansions and runtime is K-FS(100).

With suboptimality bound $w = 4.0$, WA* cannot solve any instance, and DPS can solve only one. For that reason, in this table, DPS was not taken into account to select instances solved by all algorithms.

On the other hand, FS (seq) and K-FS(1) obtain 81% and 86% of coverage, respectively. As k increases, K-FS shows better performance, achieving 100% coverage and finding better cost solutions. With this suboptimality bound, the best performance in terms of expansions and runtime was obtained with $k = 25$, performing one order of magnitude fewer expansions than FS (seq). In this domain, the time spent computing the heuristic value per state was reduced as k increases, but to a lesser extent than in the puzzle domain. In Rubik's Cube domain, one expansion generates twelve successors that can be enough to make a batch that exploits the GPU's batch processing.

Figure 2 shows the coverage vs. runtime per each algorithm. With suboptimality bound $w = 2.5$, we observe that K-FS with smaller K can quickly solve a part of the problems and slightly improves with time. As k increases, it can be slower to return a solution, but finally, it can solve most instances, such as K-FS(960), which solves the complete problem set. We observe a similar behavior with $w = 4.0$, but in this case, K-FS(25) is the fastest and solves all instances.

In summary, K-FS can reduce the time spent calculating the learned heuristic up to three orders of magnitude, increasing the number of solved instances and obtaining better cost results. However, the performance of K-FS depends on the value of k and w, which suggests that calibration of the k parameter is required. In general, with higher values of k, the number of expansions cycles will reduce. But an excessively high k value can increase the runtime and the number of expansions because the algorithm will expand a large segment of the `FOCAL` list in each expansion cycle.

VI. SUMMARY AND CONCLUSION

In this paper, we presented K-FS, a generalization of Focal Search which expands k states from `FOCAL` at every expansion cycle. The algorithm builds a batch of states to compute the learned heuristic value of a number of states in parallel exploiting the batch processing capability of a GPU. Theoretically, we prove that K-FS is complete and w-optimal and other properties of the K-FS that are related to those of KBFS. On the experimental side, we demonstrate the effectiveness of our algorithm in two classical domains, the 24-puzzle, and the Rubik's Cube, using DeepCubeA, a very effective inadmissible learned heuristic. We show that our approach outperforms others bounded-suboptimal heuristic search algorithms such as WA* and DPS and FS using the learned heuristic by two orders of magnitude in the number of expansions and three orders of magnitude in the time spent computing the heuristic. As the k value increases, the number of expansions cycles decrease, but the number of expansions may increase. This decrease in the number of expansion cycles suggests that K-FS does explore different sections of the state space compared to FS. We believe that the expansion strategy introduces more diversity to the search. However, a deeper analysis of this behavior is left for future work.

We also observe that the performance of K-FS depends on both w and k. This suggests that machine learning approaches to calibrate the k parameter may be of practical relevance, in a similar way that the job done by [29].

We also observed that K-FS is a general approach and is not restricted to learned heuristics. This paper focuses on a learned heuristic to take advantage of the batched computation on a GPU. Still, as future work we propose to explore the benefits of K-FS with a well-informed inadmissible calculated heuristic, such as FF [30].

As future work, we seek to move this approach to a concurrent algorithm that can generate the successors and explore the different zones of the state space in parallel, like PRA* [12] does.

REFERENCES

[1] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. SSC-4, no. 2, pp. 100–107, Jul. 1968. [Online]. Available: https://ieeexplore.ieee.org/document/4082128

[2] M. Helmert, "The fast downward planning system," *J. Artif. Intell. Res.*, vol. 26, pp. 191–246, Jul. 2006.

[3] G. Roger and M. Helmert, "The more, the merrier: Combining heuristic estimators for satisficing planning," in *Proc. 20th Int. Conf. Automated Planning Scheduling*, 2010, pp. 246–249.

[4] S. Toyer, S. Thiébaux, F. Trevizan, and L. Xie, "ASNets: Deep learning for generalised planning," *J. Artif. Intell. Res.*, vol. 68, pp. 1–68, May 2020.

[5] W. Shen, F. W. Trevizan, S. Toyer, S. Thiebaux, and L. Xie, "Guiding search with generalized policies for probabilistic planning," in *Proc. 12th Symp. Combinat. Search*, 2019, pp. 97–105.

[6] F. Agostinelli, S. McAleer, A. Shmakov, and P. Baldi, "Solving the Rubik's cube with deep reinforcement learning and search," *Nature Mach. Intell.*, vol. 1, no. 8, pp. 356–363, Jul. 2019.

[7] T. Li, R. Chen, B. Mavrin, N. R. Sturtevant, D. Nadav, and A. Felner, "Optimal search with neural networks: Challenges and approaches," in *Proc. 15th Int. Symp. Combinat. Search*, 2022, pp. 109–117.

[8] J. Pearl and J. H. Kim, "Studies in semi-admissible heuristics," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-4, no. 4, pp. 392–399, Jul. 1982.

[9] A. Felner, S. Kraus, and R. E. Korf, "KBFS: K-best-first search," *Ann. Math. Artif. Intell.*, vol. 39, no. 1, pp. 19–39, 2003.

[10] E. Burns, S. Lemons, W. Ruml, and R. Zhou, "Best-first heuristic search for multicore machines," *J. Artif. Intell. Res.*, vol. 39, pp. 689–743, Dec. 2010.

[11] Y. Zhou and J. Zeng, "Massively parallel A* search on a GPU," in *Proc. 29th AAAI Conf. Artif. Intell.*, 2015, pp. 1248–1255.

[12] M. Evett, J. Hendler, A. Mahanti, and D. Nau, "PRA: Massively parallel heuristic search," *J. Parallel Distrib. Comput.*, vol. 25, no. 2, pp. 133–143, Mar. 1995.

[13] S. Mukherjee, S. Aine, and M. Likhachev, "ePA*SE: Edge-based parallel A* for slow evaluations," in *Proc. 15th Int. Symp. Combinat. Search*, 2022, pp. 136–144.

[14] M. Phillips, M. Likhachev, and S. Koenig, "PA*SE: Parallel A* for slow expansions," in *Proc. 24th Int. Conf. Automated Planning Scheduling*, 2014, pp. 208–216.

[15] M. Spies, M. Todescato, H. Becker, P. Kesper, N. Waniek, and M. Guo, "Bounded suboptimal search with learned heuristics for multi-agent systems," in *Proc. 33rd AAAI Conf. Artif. Intell.*, 2019, pp. 2387–2394.

[16] P. Araneda, M. Greco, and J. A. Baier, "Exploiting learned policies in focal search," in *Proc. 14th Int. Symp. Combinat. Search*, 2021, pp. 2–10.

[17] M. Greco, J. Toro, C. Hernández-Ulloa, and J. A. Baier, "K-focal search for slow learned heuristics (extended abstract)," in *Proc. Int. Symp. Combinat. Search*, Jul. 2022, vol. 15, no. 1, pp. 279–281.

[18] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[19] P. Ferber, M. Helmert, and J. Hoffmann, "Reinforcement learning for planning heuristics," in *Proc. 1st Workshop Bridging Gap Between AI Planning Reinforcement Learn. (PRL)*, 2020, pp. 119–126.

[20] E. Groshev, M. Goldstein, A. Tamar, S. Srivastava, and P. Abbeel, "Learning generalized reactive policies using deep neural networks," in *Proc. 28th Int. Conf. Automated Planning Scheduling*, 2018, pp. 408–416.

[21] P. Ferber, M. Helmert, and J. Hoffmann, "Neural network heuristics for classical planning: A study of hyperparameter space," in *Proc. 24th Eur. Conf.*, vol. 325, 2020, pp. 2346–2353.

[22] D. Gilon, A. Felner, and R. Stern, "Dynamic potential search—A new bounded suboptimal search," in *Proc. 9th Annu. Symp. Combinat. Search*, 2016, pp. 36–44.

[23] R. T. Stern, R. Puzis, and A. Felner, "Potential search: A bounded-cost search algorithm," in *Proc. 21st Int. Conf. Automated Planning Scheduling*, 2011, pp. 234–241.

[24] J. T. Thayer and W. Ruml, "Bounded suboptimal search: A direct approach using inadmissible estimates," in *Proc. 22nd Int. Joint Conf. AI*, 2011, pp. 674–679.

[25] S. Mukherjee and M. Likhachev, "GePA*SE: Generalized edge-based parallel A* for slow evaluations," in *Proc. 16th Int. Symp. Combinat. Search*, Prague, Czech Republic, 2023, pp. 153–157.

[26] O. Hansson, A. Mayer, and M. Yung, "Criticizing solutions to relaxed models yields powerful admissible heuristics," *Inf. Sci.*, vol. 63, no. 3, pp. 207–227, Sep. 1992.

[27] R. E. Korf and A. Felner, "Disjoint pattern database heuristics," *Artif. Intell.*, vol. 134, nos. 1–2, pp. 9–22, Jan. 2002.

[28] R. E. Korf, "Finding optimal solutions to Rubik's cube using pattern databases," in *Proc. 14th National Conf. Artif. Intell.*, 1997, pp. 700–705.

[29] C. Fawcett, M. Vallati, F. Hutter, J. Hoffmann, H. H. Hoos, and K. Leyton-Brown, "Improved features for runtime prediction of domain-independent planners," in *Proc. 24th Int. Conf. Automated Planning Scheduling*, S. A. Chien, M. B. Do, A. Fern, and W. Ruml, Eds. 2014, pp. 355–359.

[30] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *J. Artif. Intell. Res.*, vol. 14, pp. 253–302, May 2001.

MATIAS GRECO received the bachelor's and master's degrees in computer science from Universidad Andrés Bello, Chile. He is currently pursuing the Ph.D. degree with Pontificia Universidad Católica de Chile.

He is the Director of the Bachelor Degree Program in Informatics Civil Engineering, Universidad San Sebastián. His research interests include artificial intelligence, including machine learning, heuristic search, automated planning, and neurosymbolic AI.

JORGE TORO received the bachelor's degree in engineering sciences from Pontificia Universidad Católica de Chile, where he is currently pursuing the degree in civil computer engineering. His research interests include artificial intelligence, natural language processing, and machine learning.

CARLOS HERNÁNDEZ received the bachelor's degree from Universidad de Concepción, Chile, and the Ph.D. degree in computer science from Universidad Autónoma de Barcelona, Spain. He is currently a Full Professor with the Faculty of Faculty of Engineering, Architecture and Design, Universidad San Sebastián. His research interests include heuristic search, automated planning, and knowledge representation, with a focus on real-time, on-line, and multi-objective problems. He is the current President of the Chilean Association of Computer Science (SCCC).

JORGE A. BAIER received the bachelor's and master's degrees from Pontificia Universidad Católica de Chile (PUC), Chile, and the Ph.D. degree in computer science from the University of Toronto, Canada. He is currently an Associate Professor with the Department of Computer Science, PUC, and the Associate Dean of Engineering Education with the School of Engineering, PUC. His research interests include automated planning, heuristic search, and AI and education.

• • •