A Faster Implementation of a Sparse Distributed Memories Architecture

Alexandre Bouchard-Côté

August 27, 2004

Abstract

Sparse Distributed Memories (SDM) [3] is a linear, local function approximation architecture having the potential to scale to problems with a high dimensionality. We are testing this hypothesis with an implementation of SDM using higher dimensional versions of existing reinforcement learning (RL) problems [2]. In order to work efficiently, our implementation of SDM needs a special data structure to store and retrieve "active locations". In this paper we discuss two possibilities for this data structure: kd-tree and hash table. Our results suggest that a hash-based implementation provides superior performances.

1 Introduction to SDM architectures and data structure requirements

Let $f : C \to \mathbf{R}$ be a function we wish to evaluate (for example Q values for an RL problem), where $C \subseteq \mathbb{R}^n$. We fix a *similarity measure* $\mu : C \times C \to [0,1]$ such that $1-\mu$ is a metric on C. The current implementation also assumes that (1) C is a closed multi-interval and (2) that there is a $\vec{\beta} = (\beta[1], \cdots, \beta[n])$ such that $\mu(\vec{x}_1, \vec{x}_2) = 0$ whenever for some i the absolute value of the i-th coordinate of the difference of $\vec{x}_1$ and $\vec{x}_2$ is greater than or equal to $\beta[i]$.

In the core of an SDM architecture is a set H of pairs $(\vec{h}_k, w_k : \vec{h}_k \in C, w_k \in \mathbb{R})$ called the set of *locations* (HLs). To estimate the value at $\vec{x}$ we first find the set of *active locations*, that is the subset $H_{\vec{x}}$ of H such that $H_{\vec{x}}$ contains precisely the locations with a positive similarity measure with $\vec{x}$. This is the first operation that our data structure must support. We will see in the following sections how it can be implemented efficiently using assumption (2). Once $H_{\vec{x}}$ is found, the predicted value $\tilde{f}(\vec{x})$ of $f(\vec{x})$ is computed by:

$$\tilde{f}(\vec{x}) = \frac{\sum_{(\vec{h}_k,w_k)\in H_{\vec{x}}} \mu(\vec{h}_k,\vec{x})w_k}{\sum_{(\vec{h}_k,w_k)\in H_{\vec{x}}} \mu(\vec{h}_k,\vec{x})}$$

Upon receiving a training sample $(\vec{x}, f(\vec{x}))$ the values stored in all the active locations are updated using the standard gradient descent algorithm for linear approximations:

$$w_m := w_m + \alpha \frac{\mu(\vec{h}_m,\vec{x})}{\sum_{(\vec{h}_k,w_k)\in H_{\vec{x}}} \mu(\vec{h}_k,\vec{x})} \left[f(\vec{x}) - \tilde{f}(\vec{x}) \right], \forall (\vec{h}_m, w_m) \in H_{\vec{x}}$$

where $\alpha \in (0,1)$ is a fixed constant called the *learning rate*. We also require the data structure to support the insertion and deletion of locations for dynamic resource allocation purposes.

2 Reference implementation

The existing code, written by Bohdana Ratitch, uses the following strategy to organize the storage of the locations: each dimension i is partitioned into intervals $I_1^i, \cdots, I_m^i$ of length $\beta[i]$ (this can be done using assumption (1)). For each such interval, the sets t_j^i of locations with center in $\{\vec{y} : \vec{y}[i] \in I_j^i\}$ are maintained. On input $\vec{x}$, we find the set of intervals $\{I_{j_i}^i\}_{i=1}^n$ such that for all dimensions, $\vec{x}[i] \in I_{j_i}^i$. Since

$$P := \bigcap_{i=1}^{n} (t_{j_i-1}^i \cup t_{j_i}^i \cup t_{j_i+1}^i) \supseteq H_{\vec{x}}$$

(because of assumption (2)) and that this set P of potentially active locations is usually much smaller than H, this gives a relatively more efficient way of finding $H_{\vec{x}}$ than testing all the elements of H. Unfortunately, computing the above intersection can be time consuming and we will see that we can achieve better performances using alternative methods to store the locations.

3 Hash table

The general idea for the hash-based method is to partition the space into a finite number of hyper-rectangular cells with edges parallel to the coordinate axis and of length $\beta[i]$ in the i-th dimension. We index those cells and use this index as the key for insertion and retrieval in a hash table. In order to do that we define three auxiliary maps. First, a map $\pi : C \to \{cells\}$ that gives the cell containing the given point in the space C. Second, we assume that there is an injective map $\varphi : \{cells\} \to \{finite\_strings\}$ that gives a unique string representation for any cell, so that it can be used by standard hashing algorithms. The private method *HLsupport::computeKey(·)* in the class HLsupport.cpp (see Annex) computes the composition map $\varphi \circ \pi$ and provides additional comments on implementation details. Finally, we need a set map $\Xi : C \to 2^{\{cells\}}$ that returns, for any $\vec{x} \in C$, the set $\{\xi_i\}$ of all hyper-cubic cells such that:

$$\xi_i \cap [\vec{x}[1] - \beta[1], \vec{x}[1] + \beta[1]] \times \cdots \times [\vec{x}[n] - \beta[n], \vec{x}[n] + \beta[n]] \neq \emptyset$$

Note that the returned set has a cardinality of at most 2^n. Again, refer to the annex for additional comments on the implementation of the map $\varphi \circ \Xi$ in *HLsupport::getNeighborCubesRepresentatives(·)* where φ is applied elementwise on $\Xi(\cdot)$. With the above definitions, we can now state the algorithms in pseudo-code for insertion, retrieval and deletion of locations. From now on, *index* will refer to an identifier to a location and *state*, to a point in C.

```
INSERT($state, index$)
1  $key \leftarrow \varphi \circ \pi(state)$
2  $list \leftarrow hashMap.get(key)$
3  if $list =$ NIL
4      then $list \leftarrow new\ List$
5           $list.insert(index)$
6           $hashMap.insert(list)$
7      else $list.insert(index)$
```

```
DELETE($state, index$)
1  $key \leftarrow \varphi \circ \pi(state)$
2  $list \leftarrow hashMap.get(key)$
3  $list.delete(index)$
4  if $list.isEmpty()$
5      then $hashMap.delete(key)$
```

```
FIND-POTENTIALLY-ACTIVE-LOCATIONS($state$)
1  $listOfCubes \leftarrow \Xi(state)$
2  $returnedList \leftarrow new\ List$
3  for each $cube \in listOfCubes$
4        do $key \leftarrow \varphi(cube)$
5           $list \leftarrow hashMap.get(key)$
6           $returnedList \leftarrow returnedList \cup list$
7  return $returnedList$
```

The method FIND-POTENTIALLY-ACTIVE-LOCATIONS returns a set covering H_{state} so it is easy from that to find the set H_{state}.

These algorithms were implemented in c++ and they correspond to the code for the class HLsupport in the SDM project. The hash table used is the one designed by sgi in their free distribution of STL [1]. The first step to verify correctness was to run low-level tests on the different methods constituting the class HLsupport. Those tests can be found in the file HLsupport.cpp. The second step was to fix the seed for the randomized functions in SDM's code and compare the output of the new implementation against the reference implementation. The fact that they behaved identically on both a supervised learning task and on the mountain-car domain suggests that the new implementation introduces no new bugs in the source code. An additional test to do would be to use a memory leak detector on the current code to insure that it is free of this type of problem.

In terms of speed, the delete and insert operations are not problematic since they only involve the insertion/deletion of the center of the location in a constant number of data structures. The speed bottleneck for the retrieval operation in both data structures is the number of locations processed by the returned List in its life cycle. First, each element in the list of potentially active locations returned by *HLsupport::potentiallyActiveSet(·)* will be used individually so it gives already $O(N' \cdot 3^n)$ and $O(N' \cdot 2^n)$ *List::find(·)*'s for respectively the old and new implementations, where N' is the maximum number of activated locations targeted by the SDM architecture. In the case of the new implementation this is an upper bound since the other operations (in particular, searching the hash table) are constant. On the other hand, the calls of *List::listIntersection(·)* in the reference implementation can be costly, up to $O(\max_i\{\frac{L[i]}{\beta[i]} \cdot 3N'\})$ where $L[i]$ is the length of the space along the i-th dimension. However, since the actual distribution of the locations is not uniform in general in SDMs, it is hard to evaluate an expected running time, unless a particular problem instance is fixed. This observation motivates the use of empirical methods to estimate and compare the expected running times. Our first benchmark was the mountain-car domain, in which the new implementation outperformed the old one both asymptotically and in every test. More details on the results and parameters of this benchmark are presented in the annex. The current candidate for the next benchmark is the hunter-prey domain, because it can be made more memory-intensive easily by increasing the number of hunters.

4 kd-tree

TODO

5 References

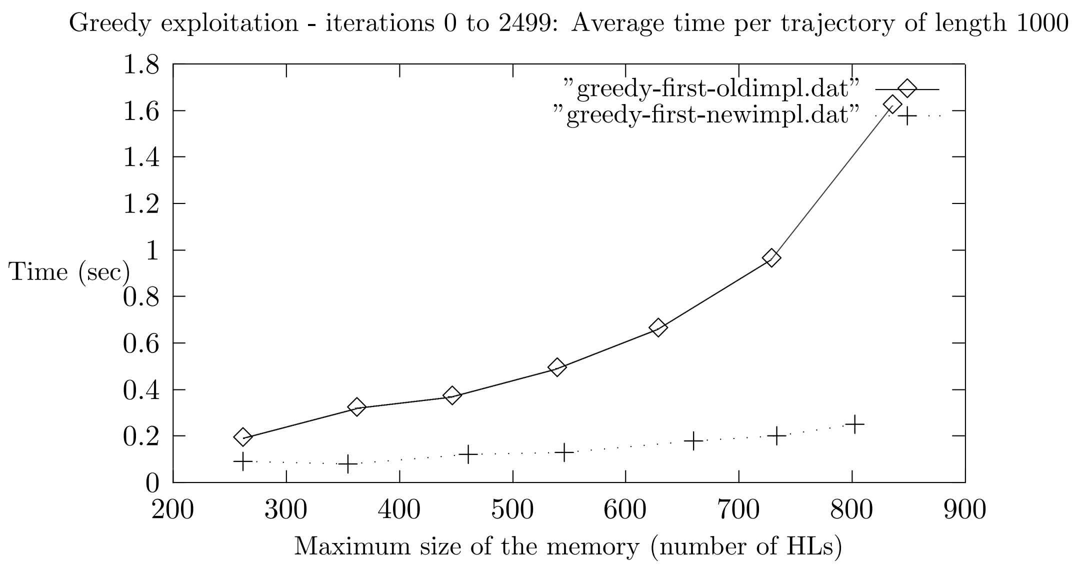
References

[1] Standard template library programmer's guide. http://www.sgi.com/tech/stl/index.html.

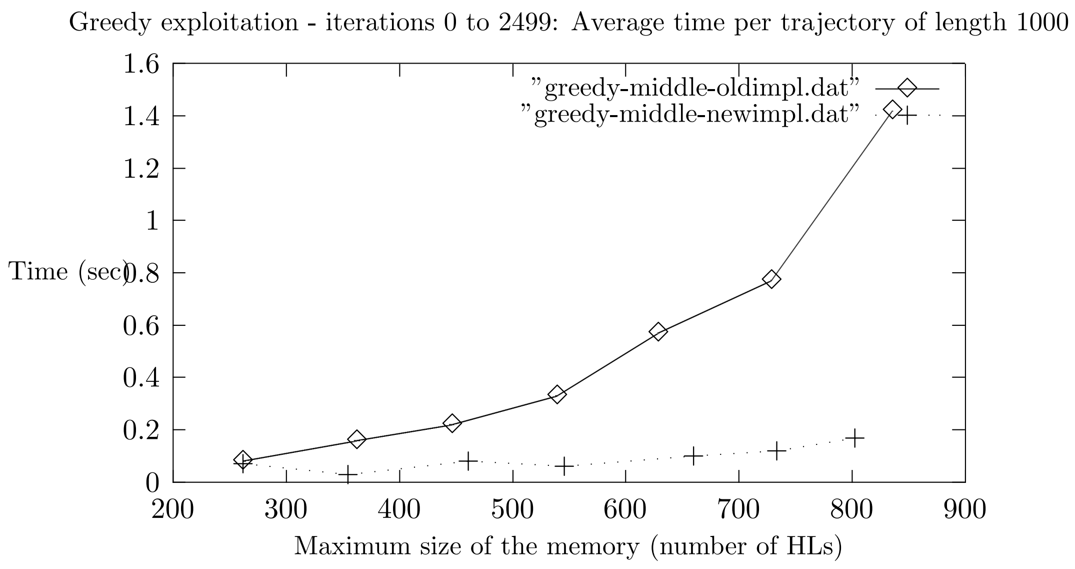
[2] D. Precup B. Ratitch. Sparse distributed memories fo on-line, value-based reinforcement learning. *ECML*, 2004.

[3] P. Kanerva. Sparse distributed memory and related models. *Associative Neural Memories*, pages 50–76, 1993.

6 Annex: Statistics and source code

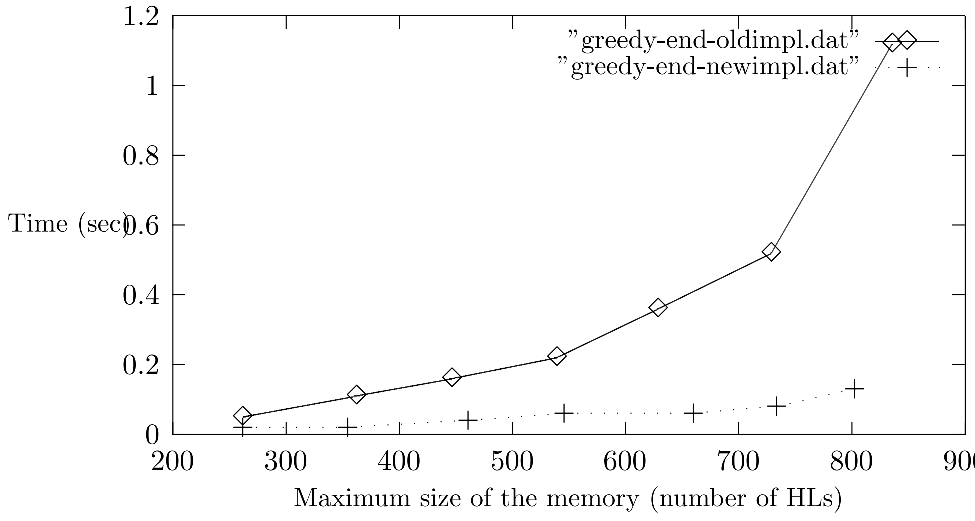


Greedy exploitation - iterations 0 to 2499: Average time per trajectory of length 1000
"greedy-first-oldimpl.dat"
"greedy-first-newimpl.dat"
Time (sec)
Maximum size of the memory (number of HLs)
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
200
300
400
500
600
700
800
900

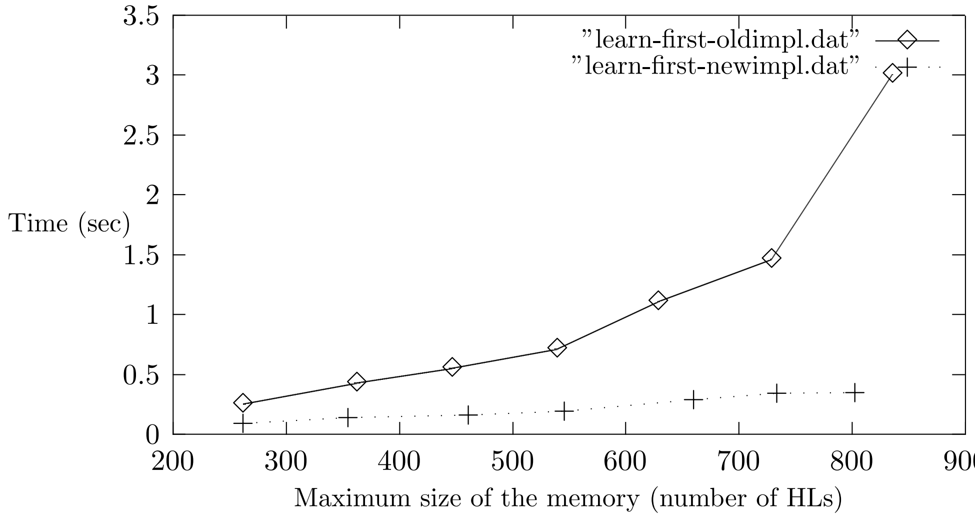


Greedy exploitation - iterations 0 to 2499: Average time per trajectory of length 1000
"greedy-middle-oldimpl.dat"
"greedy-middle-newimpl.dat"
Time (sec)
Maximum size of the memory (number of HLs)
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
200
300
400
500
600
700
800
900

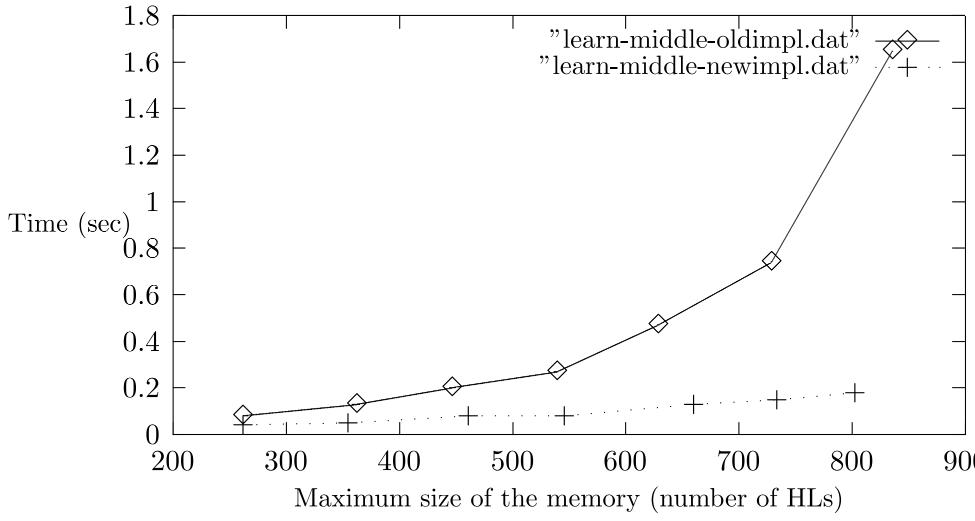
Greedy exploitation - iterations 5000 to 10000: Average time per trajectory of length 1000



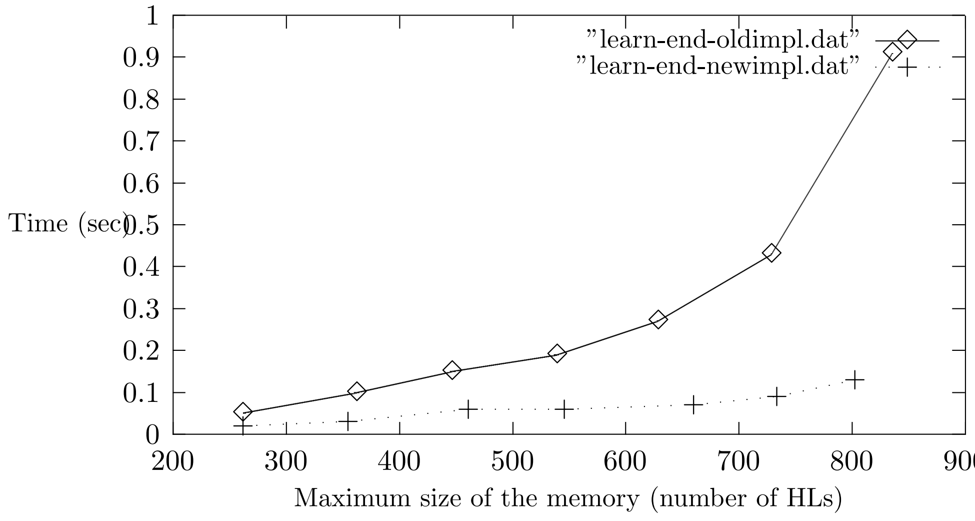
Learning - iterations 0 to 2499: Average time per trajectory of length 1000



Learning - iterations 2500 to 4999: Average time per trajectory of length 1000



Learning - iterations 5000 to 10000: Average time per trajectory of length 1000


"learn-end-oldimpl.dat"
"learn-end-newimpl.dat"
Time (sec)
Maximum size of the memory (number of HLs)