The 35th Canadian Conference on Artificial Intelligence
DOI: 10.21428/594757db.0e910d58

Canadian AI 2022
30 May–3 June 2022, Toronto

# Cache-Efficient Memory Representation of Markov Decision Processes

Jaël Champagne Gareau[†,*], Éric Beaudry[†], Vladimir Makarenkov[†]
[†] Université du Québec à Montréal

**Abstract**

Research in automated planning typically focuses on the development of new or improved algorithms. Yet, an equally important but often overlooked topic is that of how to actually implement these algorithms efficiently. In this study, we are making an attempt to close this gap in the context of optimal Markov Decision Process (MDP) planning. Precisely, we present a novel cache-efficient memory representation of MDPs, which we call CSR-MDP, that takes advantage of low-level hardware features such as memory hierarchy. We evaluate the speed improvement provided by our memory representation by comparing the performance of CSR-MDP with the performance obtained by traditional MDP representation. We show that by using our CSR-MDP memory representation, existing MDP solvers, including VI, LRTDP and TVI, are able to find an optimal policy an order of magnitude faster.

**Keywords:** Markov Decision Process, Automated Planning, High-Performance Computing, Efficient Implementation, Cache Memory

## 1. **Introduction**

Markov Decision Processes (MDPs) are used to model problems of decision-making under uncertainty. These problems involve an agent that needs to achieve an objective by executing optimal (or nearly optimal) actions among a set of applicable ones. In automated planning, a priori knowledge of a probabilistic model of the actions' effects is assumed [1]. MDPs are also used in (Model-Based) Reinforcement Learning (RL), but in this context, a probabilistic model is typically assumed to be unknown and must be learned either through real-world experiments or through simulated experiments (using a given sampling model) [2].

Once an MDP model is known (through a priori knowledge or through learning), one generally wants to find an optimal policy, i.e., a mapping specifying which action should be executed in each state to achieve an objective with maximum reward (or minimum cost). In automated planning, dynamic programming algorithms such as Value Iteration (VI) [3] and Policy Iteration [4], are generally used to find such a policy.

Recent progresses made in planning algorithms to solve MDPs faster involve heuristic search algorithms and trial-based (sampling) algorithms. Labeled Real-Time Dynamic-Programming (LRTDP) [5] and LAO* [6] are examples of planning algorithms combining both of these ideas.

An orthogonal way to improve the running time of MDP solvers is the exploitation of advanced features and architectures of modern CPUs, including cache memory and vector (Single Instruction Multiple Data, SIMD) instructions (e.g., SSE or AVX instructions on x86 processors). Adapting existing MDP solvers to the use of the aforementioned elements can lead to substantial performance improvement, as can be seen in many problems studied by the High-Performance Computing (HPC) research community [7–9]. Over the last few years, Machine Learning (ML) algorithms have benefited from substantial performance gains due to the consideration of low-level computer architecture elements. For example, specialized floating point numbers, such as bfloat16, and specialized SIMD instructions using these number types allowed a significant speedup in many ML computations [10, 11]. Parallelism,

[*]champagne_gareau.jael@uqam.ca

achieved either through CPU or GPU, and memory techniques, such as tiling, also helped many ML algorithms solve much larger classification problems [12–14]. Since the described techniques allowed considerable performance gains in ML, one can expect that similar ideas applied to AI planning could lead to MDP solvers capable of tackling efficiently larger real-world problems than currently possible.

In this paper, we show that by exploiting the memory hierarchy of modern computers, state-of-the-art MDP solvers can run an order of magnitude faster. Our main contributions are as follows: (1) we present a novel cache-efficient memory representation of MDP, which we call CSR-MDP, and (2) we evaluate the performance of CSR-MDP on 3 different MDP domains, comparing it to a traditional MDP representation when using the VI, LRTDP and Topological Value Iteration (TVI) algorithms.

The remainder of the paper is structured as follows: Sections 2 and 3 respectively present a quick survey of existing MDP solvers and formally define MDPs and other concepts used in our study. Section 4 introduces our cache-efficient MDP memory representation, i.e., CSR-MDP. We finally present our empirical evaluation in Section 5, and conclude in Section 6.

## 2. Related Work

Many MDP solvers are based on the Value Iteration (VI) algorithm [3], or more precisely on asynchronous variants of VI. In asynchronous VI, MDP states can be backed up in any order and don't need to be considered the same number of times. One way to take advantage of this is by assigning a priority to every state and considering them in priority order. Prioritized Sweeping [15] is an example of an algorithm using this idea. However, the cost of maintaining the priority queue used to control the states backup order often cancels the potential speedup. One way to reduce this cost is to divide the states into partitions and then to assign a priority to these partitions (instead of assigning a priority to states). The General Prioritized Solvers (GPS) family of algorithms use such a strategy. GPS algorithms are able to achieve two orders of magnitude speed improvement on many MDP domains when compared to Prioritized Sweeping [16]. One downside of GPS is that there is no general method for partitioning the states. Hence, an efficient partitioning must be found according to specific features of the MDP domain of interest (e.g., if states represent locations, a k-means-like partitioning can be carried out).

More recently, a general way of partitioning MDP states has been proposed for the Topological Value Iteration (TVI) algorithm [17]. TVI considers the graphical structure of the MDP (equivalently, the structure of the graph resulting from the all-outcome determinization of the MDP) and uses the Kosaraju algorithm to find its strongly connected components (SCCs). TVI then runs VI on every SCC in reverse topological order. Since, by definition, there are no cycles between SCCs, this order is optimal and every SCCs must be considered only once [18]. TVI is orders of magnitude faster than state-of-the-art general MDP solvers, such as LRTDP [5], BRTDP [19] and ILAO* [6], on domains containing many SCCs. Any MDP domain containing state variables that can only change monotonically (only increase or only decrease) will have many SCCs and be a good candidate for TVI. For example, board games like chess have many SCCs since the total number of pieces of a given player can never grow. One disadvantage of TVI is that SCCs can sometimes be huge (in the worst case, an MDP includes only one SCC containing every state), and solving these SCCs with VI can take a while. To alleviate this problem, the Focused Topological Value Iteration (FTVI) algorithm uses a heuristic search to quickly find sub-optimal actions [17]. These actions are then pruned from the MDP, sometimes allowing more SCCs to be found. However, FTVI requires lower and upper-bound heuristics and the algorithm performance greatly depends on their informativeness.

To the best of our knowledge, only one study considers (CPU) cache performance of MDP solvers [20]. The proposed algorithm, called Cache-Efficient with Clustering (CEC), subdivides the SCCs found by the FTVI algorithm into groups of states (or "clusters") of a size that fits the L3 CPU cache memory. The step of FTVI consisting in solving an SCC using VI is transformed into a procedure that cyclically considers (i.e. solves) every cluster in the SCC until the entire SCC converges. The authors indicated that their algorithm allowed them to achieve a speedup factor varying between 2 and 8 compared to FTVI. However, as we were able to observe by studying the source code of their CEC implementation, the data structures they used (a linked list of linked lists) are not optimal for memory accesses, probably causing an overestimate of the realistic achievable performance gains provided by CEC.

Other works have considered memory caches of hard drives when MDP instances don't fit totally in the main memory [21], but we don't discuss them here since the problem in this case is somewhat orthogonal to our research.

## 3. Problem Definition

There exist different types of MDP, including Finite-Horizon MDP, Infinite-Horizon MDP and Stochastic Shortest Path MDP (SSP-MDP) [1]. The first two of them can be seen as special cases of SSP-MDP [18]. In this work, we focus on SSP-MDPs, which we describe formally in Definition 1 below.

**Definition 1.** A *Stochastic Shortest Path MDP* (SSP-MDP) is a tuple $(S, A, T, C, G)$, where:

- $S$ is a finite set of (discrete) states;
- $A$ is a finite set of actions;
- $T\colon S \times A \times S \to [0,1]$ is a transition function, where $T(s,a,s')$ is the probability of reaching state $s'$ when applying action $a$ while in state $s$;
- $C\colon S \times A \to \mathbb{R}^+$ is a cost function, where $C(s,a)$ gives the cost of applying the action $a$ while in state $s$;
- $G \subseteq S$ is the set of goal states (which can be assumed to be sink states).

We generally look for a policy $\pi\colon S \to A$, indicating which action should be executed at each state, such that an execution starting at any state and following the actions given by $\pi$ until a goal is reached has a minimal expected cost. The expected cost of following a policy $\pi$ when starting at a specific state is given by a value function $V^\pi\colon S \to \mathbb{R}$. The Bellman Optimality Equations (Definition 2) are a system of equations satisfied by any optimal policy.

**Definition 2.** The Bellman Optimality Equations are the following:

$$V(s) = \begin{cases} 0, & \text{if } s \in G \\ \min_{a \in A} \left[ C(s,a) + \sum_{s' \in S} T(s,a,s')V(s) \right] & \text{otherwise.} \end{cases}$$

The part between square brackets is called the Q-value of a state-action pair:

$$Q(s,a) = C(s,a) + \sum_{s' \in S} T(s,a,s')V(s).$$

When an optimal value function $V^\star$ is known, an optimal policy $\pi^\star$ can be found greedily:

$$\pi^\star(s) = \operatorname*{argmin}_{a \in A} Q^\star(s,a).$$

Most MDP solvers use dynamic programming algorithms like Value Iteration (VI), which update iteratively an arbitrarily initialized value function until convergence with a given

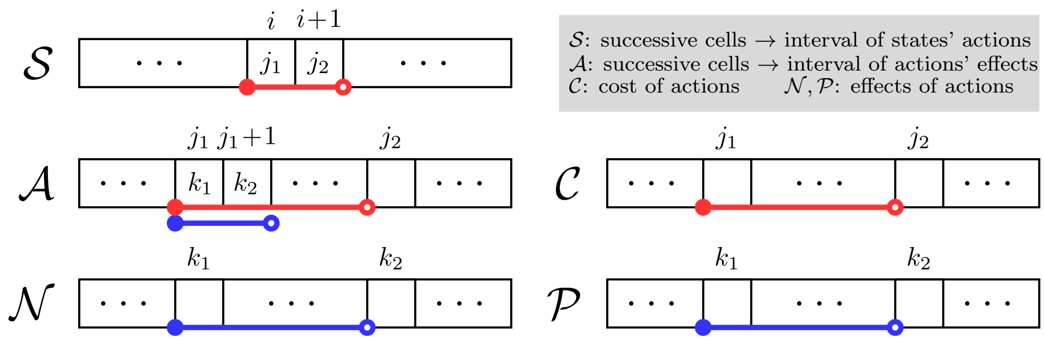



*Figure 1.* CSR-MDP memory representation scheme

precision $\epsilon$. In the worst case, VI needs to do $|S|$ sweeps of the state space, where one sweep consists in updating the value estimate of every state using the Bellman Optimality Equations. Hence, the number of state updates (called a *backup*) is $\mathcal{O}(|S|^2)$. When the MDP is acyclic, most of these backups are wasteful, since the MDP can in this situation be solved using only $|S|$ backups (ordered in reverse topological order), thus allowing one to find an optimal policy in $\mathcal{O}(|S|)$ [18].

## 4. Memory Representation

Research in this domain generally focuses on theoretical advances like heuristic search (e.g., LRTDP, LAO*, etc.). The implementation details get much less attention. We argue that the choice of the memory representation used to store an MDP can have a significant impact on the MDP solver performance, which is sometimes even more important than the choice of the solver per se (e.g., VI vs. LRTDP). This difference in performance is mostly due to the CPU cache performance (e.g., cache hit rate) of the data structures being used, which varies greatly among them (e.g., arrays and linked lists have totally different cache access patterns).

By analyzing the source code of different publicly available MDP implementations, we could get an idea about the most common data structures used. AI Toolbox, a popular MDP and POMDP C++ library [22], lets the user choose between dense or sparse matrices to store MDPs (one 3D matrix of transitions and one 2D matrix for costs/rewards). The dense matrices almost always take an unreasonable amount of memory, even on small MDPs. On the other hand, sparse matrices are implemented in such a way that a minimal amount of memory is wasted, but at the cost of some possible decrease in computational speed. In the implementations of TVI, FTVI and CEC by their original authors, MDP states are represented by a linked list of states, each containing a linked list of applicable action effects. Other implementations, such as Blai Bonet's MDP-Engine library[1] or Gourmand's implementation G-Pack [23], use hash tables of structures to store the MDP states. All of these implementations use a representation that we could classify as an "Array of Structures" (AoS) memory layout representation (on the opposite to the "Structure of Arrays" memory layout), and none of them explicitly stores the MDP in a cache-optimal way.

Modern CPUs have multiple levels of cache memory, usually named L1, L2 and L3, where L1 is the smallest and the fastest cache, and L3 is the slowest but the largest cache. These cache memories allow the computer to load recently used data without having to wait for central memory. The smallest amount of data loaded at a time in memory, named the cache

[1] https://github.com/bonetblai/mdp-engine

line size, is usually 64 bytes on modern CPUs. Access to the fastest level of cache is usually around 3 orders of magnitude faster than access to central memory.

There are two main ways of taking advantage of cache memory and thus decreasing the memory access time: (1) data that are often used together can be packed in memory to ensure that all memory inside loaded cache lines is useful for the current computation; (2) algorithms can be modified to minimize the amount of memory accesses, e.g., by working longer with loaded data before loading different data.

In this section, we present a novel memory representation of MDP which, to the best of our knowledge, has never been described before. This representation is inspired by the Compressed Sparse Row (CSR) representation of directed graphs [24], known to yield excellent cache performance with minimal memory overhead [25]. We can classify our CSR-MDP representation as a "Structure of Arrays" (SoA) memory layout representation.

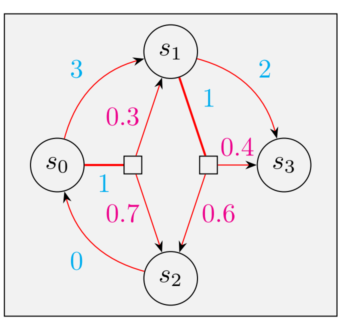
Figure 1 illustrates our MDP memory representation scheme, which we call CSR-MDP. This representation includes five arrays, $(\mathcal{S}, \mathcal{C}, \mathcal{A}, \mathcal{N}, \mathcal{P})$, where:

- $[\mathcal{S}[i],\ \mathcal{S}[i+1][$ is the interval of indices of state $i$'s actions;
- $\mathcal{C}[j]$ is the cost of executing action $j$;
- $[\mathcal{A}[j],\ \mathcal{A}[j+1][$ is the interval of indices of action $j$'s probabilistic effects;
- $\mathcal{N}[k]$ is the id of the state reached when the effect $k$ of an action occurs;
- $\mathcal{P}[k]$ is the probability that the effect $k$ of an action occurs.

In Figure 1, the red lines under $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{C}$ symbolize data associated with state $i$. The two numbers in the red region in $\mathcal{S}$ represent the semi-open interval of indices in $\mathcal{A}$ and $\mathcal{C}$ corresponding respectively to state $i$'s actions' effects and costs. Similarly, the blue lines under $\mathcal{A}$, $\mathcal{N}$ and $\mathcal{P}$ symbolize data associated to action $j_1$. The two numbers in the blue region in $\mathcal{A}$ represent the semi-open interval of indices in $\mathcal{N}$ and $\mathcal{P}$ corresponding to action $j_1$'s possible outcomes (possible neighbors and respective transition probabilities).

The arrays $\mathcal{C}$ and $\mathcal{A}$, and the arrays $\mathcal{N}$ and $\mathcal{P}$, can be respectively merged into single arrays of pairs of variables, but we chose not to do it in our implementation because: (1) we don't always need to access both variables at the same time (and in such a case, we can double the amount of useful information in the cache by keeping the arrays separate), and (2) the eventual use of SIMD instructions in an optimized solver would require (or at least benefit from) contiguous actions' cost data or actions' probabilistic effects in memory.

Figure 2 shows an example of an SSP-MDP and its associated CSR-MDP representation. The numbers in cyan represent the cost of each action. Only one action in state 0 and one action in state 1 are non-deterministic. The numbers in magenta represent the probability of the outcomes of these actions.




$\mathcal{S}$

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 0 | 2 | 4 | 5 | 5 |

$\mathcal{A}$

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 0 | 1 | 3 | 4 | 6 | 7 |

$\mathcal{C}$

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 3 | 1 | 2 | 1 | 0 |

$\mathcal{N}$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 3 | 2 | 0 |

$\mathcal{P}$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0.3 | 0.7 | 1 | 0.4 | 0.6 | 1 |

*Figure 2.* An example of an MDP and its corresponding CSR-MDP representation

The main advantage of CSR-MDP over other MDP memory representations is that in CSR-MDP all data are packed together, maximizing therefore the cache efficiency. Another advantage of CSR-MDP is that data are stored homogeneously in separate arrays, making it easier for the programmer or the compiler to vectorize the code using SIMD instructions. Memory-wise, CSR-MDP has smaller overhead compared to the existing representations. For example, with linked lists, a significant part of memory is used to store the pointers between cells, while with hash tables, memory is wasted by empty buckets. Note that in our implementation, the elements of every array are stored in 4 bytes, since we only use 4-bytes integers (in $\mathcal{S}, \mathcal{A}$ and $\mathcal{N}$) or 4-bytes floating point numbers (in $\mathcal{C}$ and $\mathcal{P}$).

Suppose we have an MDP $M$ containing $n$ states, where the average number of applicable actions per state is $m$ and the average number of probabilistic effects per action is $k$. The total memory (in bytes) necessary to store $M$ entirely in the CSR-MDP data structure can be assessed as follows:

$$\begin{aligned}
MemorySize(M) &= 4(n+1) && (\mathcal{S} \text{ array})\\
&+ 4(nm) && (\mathcal{C} \text{ array})\\
&+ 4(nm+1) && (\mathcal{A} \text{ array})\\
&+ 4(nmk) && (\mathcal{N} \text{ array})\\
&+ 4(nmk) && (\mathcal{P} \text{ array})\\
&= 8nm(k+1) + 4n + 8. && (\text{in bytes})
\end{aligned}$$

For example, to store an instance of the Single-Armed Pendulum problem (described in Section 5, where $m = 2$ and $k = 3$) containing $n$ states would require $68n + 8$ bytes.

## 5. Empirical Evaluation

In this section, we evaluate the performance of the CSR-MDP memory representation. To do so, we compare the performance of our implementation (which uses CSR-MDP) to the performance of a baseline implementation. As a baseline, we used the implementation used for the evaluation in TVI's and CEC's original paper [17, 20]. The memory representation used in the baseline is an Array of Structures (AoS) representation. More specifically, it consists of a linked-list of pointers to structures representing each state. This baseline is representative of the public MDP implementations available, which, to the best of our knowledge, all use an AoS MDP representation.

We compare the CSR-MDP and baseline memory representations by assessing their impact on the performance of the three following algorithms: (1) VI – the standard dynamic programming algorithm (we use the asynchroneous round-robin variant), (2) LRTDP – a well-known heuristic search algorithm, and (3) TVI – the Topological Value Iteration algorithm described in Section 2. For LRTDP, we used the admissible and domain-independent $h_{\min}$ heuristic, first described in the original paper introducing LRTDP [5]:

$$h_{\min}(s) = \begin{cases} 0, & \text{if } s \in G. \\ \min\limits_{a \in A_s} \left[ C(s,a) + \min\limits_{s' \in succ_a(s)} V(s') \right], & \text{otherwise,} \end{cases}$$

where $A_s$ denotes the set of applicable actions in state $s$, and $succ_a(s)$ is the set of successors when applying action $a$ at state $s$. The three competing algorithms (VI, LRTDP and TVI) were implemented in C++ by the authors of this paper and compiled using the GNU g++ compiler (version 11.2, with level 3 optimizations). We did not attempt to vectorize the code manually using SIMD instructions, but the compiler auto-vectorized some parts of it. All tests were performed on a PC computer equipped with a 4.2 GHz Intel Core i5-7600K Processor with 16 GB of RAM memory. For every test domain, we measured the running time of the three compared algorithms carried out until convergence to an $\epsilon$-optimal value function (the value of $\epsilon$ was fixed to $10^{-6}$ in our study). Every domain size was tested 15 times with randomly generated MDP instances. To minimize random factors, we report the median values obtained over these 15 MDP instances.

We evaluated the performance of the VI, LRTDP and TVI algorithms on 3 different MDP domains. The first of them is the generic Layered domain described in TVI's paper [17]. This domain is parameterized by four different parameters: $n, n_l, n_a$ and $n_s$, respectively describing the number of *states*, the number of *layers*, the number of *applicable actions per state*, and the maximum number of *successor states per action* (i.e., every action $a$ can lead to $k_a$ different states, where $k_a$ is drawn from a uniform integer distribution in $[1, n_s]$). Transition probabilities are uniformly sampled from possible successors. States in this domain are evenly divided into $n_l$ layers, $\{1, 2, \ldots, n_l\}$. A state in layer $i$ can only have successor states in layers $\{i+1, \ldots, n_l\}$, which means that MDPs in this layered domain have at least $n_l$ SCCs. The second domain we considered is the Single-Armed Pendulum (SAP) domain [16]. This domain represents a two-dimensional minimum-time optimal control problem in which an agent always has two possible actions: apply a positive or a negative torque to a rotating pendulum. The objective of the agent is to balance the pendulum to the top. The state space is defined by two variables: angle $\theta$ and angular velocity $\omega$. Finally, the last domain we used in our evaluation is a variant of the Wetfloor domain [26]. In this domain, the state space is a square navigation grid in which cells can be in one of three levels of wetness: dry, slightly wet or heavily wet. In the grid, cells are independently chosen as wet with probability $p$. Among wet cells, a second parameter $q$ controls the probability of being slightly wet ($q$) or heavily wet ($1 - q$). The agent starts in a certain position and the goal is to reach another position with a minimal number of actions. The actions are {Up, Down, Left, Right}. They are deterministic on dry cells. On wet cells, the actions outcome is probabilistic and depends on parameters $r_{slightly}$ and $r_{heavy}$. In our evaluation, we used a modified Wetfloor domain where instead of having a single square grid, we have many such grids connected to each other (intuitively, this represents many wet rooms in a house).

| Domain | VI | LRTDP | TVI |
|---|---|---|---|
| Layered (var. states) | 5.87481 | 7.91771 | 4.46547 |
| Layered (var. layers) | 6.77031 | > 4.07741 | > 3.87843 |
| SAP | 4.36132 | > 1.0539 | 5.34032 |
| Wetfloor | > 15.3342 | > 13.812 | 12.3605 |
| **Average** | > 8.69197 | > 2.86112 | > 6.6481 |

*Table 1.* Average speedup factors obtained by every solver on every domain using the proposed CSR-MDP data structure when compared to the baseline implementation. Numbers with the '>' symbol are lower bounds on the true speedup factor.

Table 1 reports the average speedup factors provided by CSR-MDP when compared to the baseline implementation. We compare the speedups obtained on the three tested domains with every tested solver, to see if a specific solver benefits more from the CSR-MDP representation. Table 2 presents the detailed results obtained on different instances of the considered Layered, SAP and Wetfloor domains. The first four columns report the characteristics of the considered MDP instances, including the domain name, the number of states of the generated instance, the number of SCCs (we don't count here the SCC containing only the goal state) and the size of the largest SCC. The next six columns present the median running time (in ms) obtained by the three tested algorithms (VI, LRTDP and TVI) when carried out with the baseline implementation and with our CSR-MDP implementation. Figures 3, 4, 5 and 6 illustrate the obtained results graphically. The 'b' and 'csr' subscripts in the figures denote respectively the baseline and the CSR-MDP implementations.

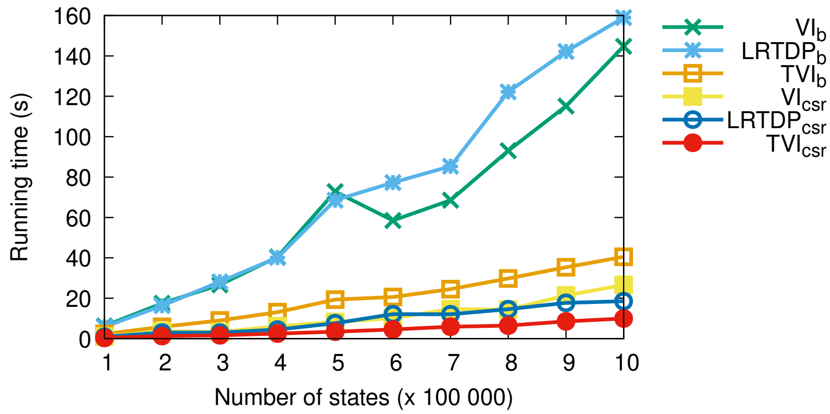



*Figure 3.* Running times (in s) for the Layered domain with varying number of states and fixed number of layers (10).

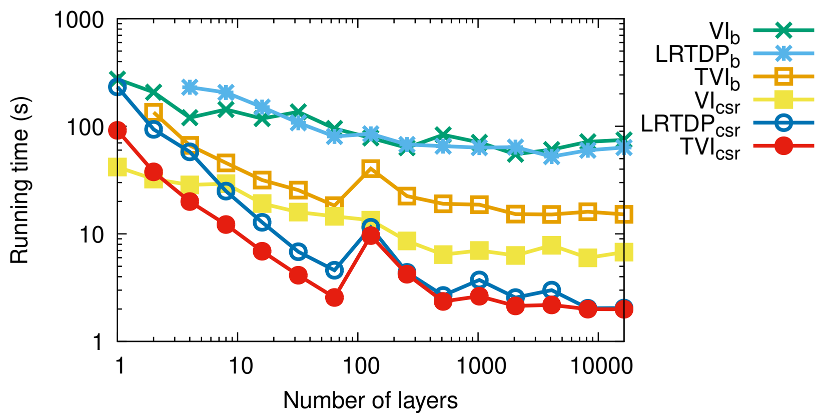



*Figure 4.* Running times (in s) for the Layered domain with varying number of layers and fixed number of states (1M). Both axes have a logarithmic scale.

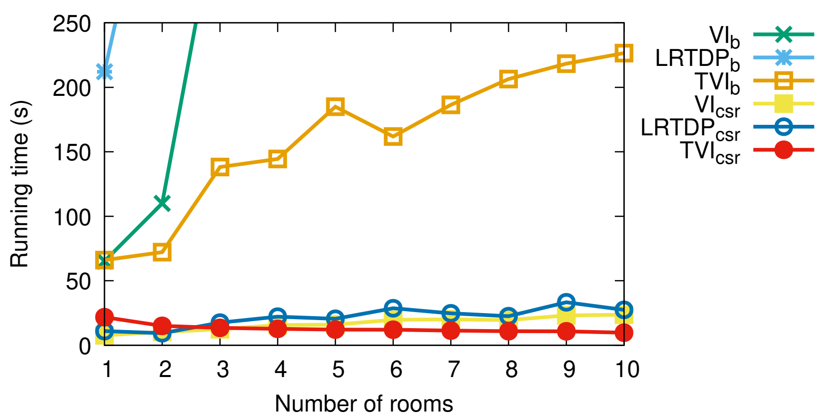



*Figure 5.* Running time (in s) for the Wetfloor domain with varying number of rooms and fixed number of states (500k).

| Characteristics of MDP instances | | | | Baseline | | | CSR-MDP | | |
|---|---|---|---|---|---|---|---|---|---|
| Domain | $\|S\|$ | $SCC$ | $\|SCC_{\max}\|$ | VI | LRTDP | TVI | VI | LRTDP | TVI |
| Layered | 100,000 | 10 | 10,000 | 6,253 | 5,829 | 2,355 | 1,051 | 905 | **400** |
| Layered | 200,000 | 10 | 20,000 | 17,499 | 16,301 | 5,828 | 3,742 | 3,018 | **1,323** |
| Layered | 300,000 | 10 | 30,000 | 26,655 | 28,131 | 9,001 | 3,243 | 3,006 | **1,633** |
| Layered | 400,000 | 10 | 40,000 | 40,489 | 40,160 | 13,142 | 6,154 | 4,491 | **2,543** |
| Layered | 500,000 | 10 | 50,000 | 72,815 | 68,544 | 19,343 | 8,262 | 7,614 | **3,447** |
| Layered | 600,000 | 10 | 60,000 | 58,511 | 77,312 | 20,602 | 10,135 | 12,103 | **4,575** |
| Layered | 700,000 | 10 | 70,000 | 68,574 | 85,359 | 24,532 | 14,606 | 12,093 | **5,928** |
| Layered | 800,000 | 10 | 80,000 | 93,054 | 122,189 | 29,821 | 14,392 | 14,601 | **6,466** |
| Layered | 900,000 | 10 | 90,000 | 115,157 | 142,215 | 35,335 | 21,385 | 17,718 | **8,591** |
| Layered | 1,000,000 | 10 | 100,000 | 144,708 | 158,977 | 40,554 | 26,602 | 18,546 | **9,997** |
| Layered | 1,000,000 | 1 | 1,000,000 | 273,731 | - | - | **41,788** | 232,539 | 91,612 |
| Layered | 1,000,000 | 2 | 500,000 | 207,160 | - | 134,678 | **32,291** | 93,500 | 37,682 |
| Layered | 1,000,000 | 4 | 250,000 | 120,241 | 230,598 | 66,380 | 28,539 | 57,884 | **19,941** |
| Layered | 1,000,000 | 8 | 125,000 | 143,112 | 206,107 | 45,595 | 29,266 | 24,954 | **12,210** |
| Layered | 1,000,000 | 16 | 62,500 | 117,847 | 150,479 | 31,583 | 19,123 | 12,806 | **6,898** |
| Layered | 1,000,000 | 32 | 31,250 | 136,257 | 107,789 | 25,469 | 15,946 | 6,814 | **4,128** |
| Layered | 1,000,000 | 64 | 15,625 | 95,319 | 80,321 | 18,271 | 14,543 | 4,592 | **2,562** |
| Layered | 1,000,000 | 128 | 7,813 | 77,791 | 84,933 | 40,361 | 13,413 | 11,500 | **9,637** |
| Layered | 1,000,000 | 256 | 3,907 | 63,078 | 67,585 | 22,438 | 8,595 | 4,391 | **4,214** |
| Layered | 1,000,000 | 512 | 1,954 | 83,238 | 65,363 | 19,015 | 6,417 | 2,676 | **2,353** |
| Layered | 1,000,000 | 1,024 | 977 | 70,765 | 63,297 | 18,698 | 7,009 | 3,732 | **2,630** |
| Layered | 1,000,000 | 2,048 | 489 | 54,964 | 63,966 | 15,267 | 6,288 | 2,559 | **2,141** |
| Layered | 1,000,000 | 4,096 | 245 | 60,675 | 52,289 | 15,141 | 7,824 | 2,995 | **2,184** |
| Layered | 1,000,000 | 8,192 | 123 | 71,687 | 59,868 | 16,050 | 5,981 | 2,035 | **1,997** |
| Layered | 1,000,000 | 16,384 | 62 | 74,595 | 63,480 | 15,198 | 6,756 | 2,043 | **1,992** |
| SAP | 10,000 | 1 | 10,000 | 117 | 343 | 121 | 32 | 227 | **12** |
| SAP | 40,000 | 1 | 40,000 | 1,155 | 8,552 | 1,170 | 234 | 4,962 | **109** |
| SAP | 90,000 | 1 | 90,000 | 3,673 | 42,597 | 3,735 | 857 | 16,184 | **476** |
| SAP | 160,000 | 1 | 160,000 | 7,663 | 142,882 | 7,702 | 1,903 | 78,565 | **1,538** |
| SAP | 250,000 | 1 | 250,000 | 15,387 | - | 15,665 | 3,919 | 292,439 | **3,124** |
| SAP | 360,000 | 1 | 360,000 | 25,262 | - | 25,292 | 6,055 | - | **4,852** |
| SAP | 490,000 | 1 | 490,000 | 39,694 | - | 39,914 | 9,591 | - | **7,425** |
| SAP | 640,000 | 1 | 640,000 | 54,786 | - | 55,188 | 13,832 | - | **11,772** |
| SAP | 810,000 | 1 | 810,000 | 81,573 | - | 81,939 | 18,756 | - | **15,707** |
| SAP | 1,000,000 | 1 | 1,000,000 | 111,414 | - | 111,861 | 22,945 | - | **19,136** |
| Wetfloor | 500,000 | 1 | 500,000 | 65,185 | 212,101 | 65,968 | **7,721** | 10,997 | 21,756 |
| Wetfloor | 500,000 | 2 | 250,000 | 109,979 | - | 72,191 | **10,297** | 9,326 | 14,988 |
| Wetfloor | 500,000 | 3 | 166,667 | - | - | 138,184 | **12,414** | 17,502 | 13,514 |
| Wetfloor | 500,000 | 4 | 125,000 | - | - | 144,481 | 15,582 | 22,166 | **12,711** |
| Wetfloor | 500,000 | 5 | 100,000 | - | - | 184,969 | 15,985 | 20,536 | **12,116** |
| Wetfloor | 500,000 | 6 | 83,334 | - | - | 161,854 | 19,680 | 28,703 | **12,075** |
| Wetfloor | 500,000 | 7 | 71,429 | - | - | 186,367 | 19,972 | 24,696 | **11,400** |
| Wetfloor | 500,000 | 8 | 62,500 | - | - | 206,437 | 19,524 | 22,522 | **10,845** |
| Wetfloor | 500,000 | 9 | 55,556 | - | - | 218,317 | 23,127 | 33,287 | **10,790** |
| Wetfloor | 500,000 | 10 | 50,000 | - | - | 226,503 | 23,634 | 27,467 | **9,676** |

*Table 2.* Median running times (in ms) found for every tested domain are indicated. Fastest time for each domain instance is bolded. The symbol '-' indicates when a solver could not solve an instance within 5 minutes.

Figure 3 presents the obtained results for the Layered domain when fixing the number of layers (10) and varying the number of states (from 100k to 1M), whereas Figure 4 shows the results for this domain when fixing the number of states (1M) and varying the number of layers (from 1 to 16384). In the first case (varying the number of states), we can see that TVI is able to leverage the 10 layers which allows it to clearly beat VI and LRDTP.

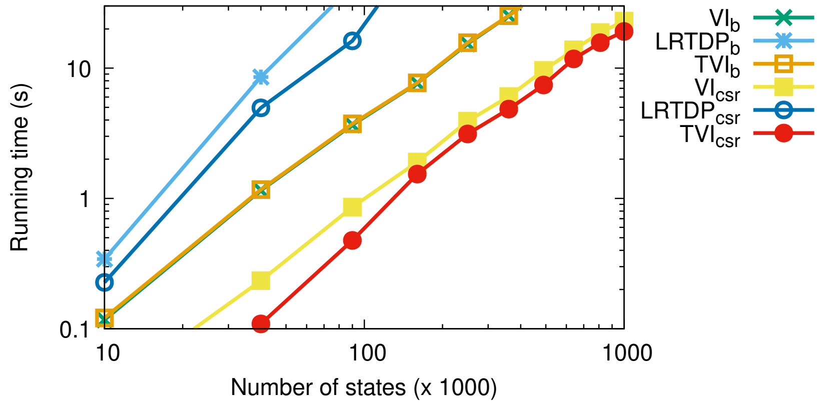



*Figure 6.* Running time (in s) for the SAP domain with varying number of states. Both axes have a logarithmic scale.

The baseline implementations are much slower than the CSR-MDP implementation in this domain. In the second case (varying the number of layers), we can additionally observe that LRTDP and TVI become slower at the level of 128 layers, which can be explained by the fact that with this number of layers, the number of states per layer (and thus the amount of memory needed to store the complete information on the states in the layer) surpasses the size of one of the three levels of cache in the CPU. The VI algorithm is not affected by this drawback since it does not considers layers. However, LRTDP is affected by it, even though it does not explicitly considers layers, because the number of layers has an impact on the search depth attained by LRTDP before it reaches a goal.

For the SAP domain, LRTDP has a lot of difficulty to find a solution in reasonable time. Two reasons can explain this fact: (1) the $h_{\min}$ heuristic is not really informative for this domain and (2) the minimum number of actions needed to reach a goal in SAP's instances is relatively high (LRTDP is known to provide better performance when the number of actions to reach a goal is small). In the baseline implementation, VI and TVI provided equivalent performance, which was expected since the SAP domain has only one SCC (all actions are reversible), and TVI basically becomes VI when there is only one SCC in the domain. Surprisingly, the CSR-MDP implementation of TVI was faster than the CSR-MDP implementation of VI, even though the computation of the single SCC (using Tarjan's algorithm) should have caused a useless overhead. This can be explained by the fact that states in the (asynchronous) VI called by TVI are backed-up in the order they were discovered by Tarjan's algorithm (instead of their original order in the input file). This yields a much better order since states are backed-up after their neighboring states, which maximizes the speed of propagation of the values in the state-space. About 50% less sweeps through the state-space, on average, were necessary before the value-function converged to a precision $\epsilon$.

Regarding the Wetfloor domain, we can see that as for the Layered domain, TVI was able to take advantage of the number of SCCs (rooms) in the domain when compared to VI and LRTDP. However, the baseline implementation had a lot of trouble for this domain which impacted VI, LRTDP, and even TVI. The loss in performance when the number of rooms increases is due to the way of storing the SCCs in memory in this implementation. More precisely, since the states contained in an SCC are not stored contiguously, the increase in the number of SCCs causes a considerable increase in the number of cache-misses. For this domain, VI's and LRTDP's performance decreases as the number of rooms increases. In the case of VI, it can be explained by the fact that when the number of rooms is high, most backups carried out by VI are useless (they propagate unconverged state-values). In the case of LRTDP, it can be explained by the fact that a higher number of rooms generally corresponds to a larger search depth from the initial state to the goal state in this domain.

## 6. Conclusion

In this paper, we presented a new way of storing an MDP in memory, called CSR-MDP, which was inspired by the Compressed Sparse Row (CSR) representation of sparse graphs. Our memory representation reduces the memory overhead induced by most other representations, while packing MDP data contiguously in memory. This strategy minimizes the memory access time when solving MDPs. The results of our experiments conducted with different probabilistic planning domains indicate that the CSR-MDP representation provides an average speedup factor (over all tested domains) of 8.6, 2.8 and 6.6, when using VI, LRTDP and TVI, respectively. Our implementations, including the domains generators, are freely available online[2].

In the future, we plan to further consider and take advantage of the cache hierarchy of modern CPUs by developing a new MDP solver that decomposes an MDP into smaller subparts which can be considered one at a time, fitting entirely in cache memory. This could almost completely eliminate the cache misses when solving a large MDP, leading to further substantial speedups. We also plan on investigating if the proposed representation can be used in GPU-based implmentations and yield further performance improvements.


## Acknowledgements

This research has been supported by the *Natural Sciences and Engineering Research Council of Canada* (NSERC) and the *Fonds de Recherche du Québec — Nature et Technologies* (FRQNT). We would also like to thank Dr. David Wingate for sending us his SAP domains generator and Anuj Jain, for providing us the original implementation of TVI.


## References


[1] Mausam and A. Kolobov. *Planning with Markov Decision Processes: An AI Perspective*. 1. Morgan & Claypool, 2012, pp. 1–210. DOI: 10.2200/S00426ED1V01Y201206AIM017.

[2] R. S. Sutton and A. G. Barto. *Reinforcement Learning, Second Edition: An Introduction*. MIT Press, 2018. ISBN: 9780262039246. URL: https://mitpress.mit.edu/books/reinforcement-learning-second-edition.

[3] R. Bellman. *Dynamic Programming*. Prentice Hall, 1957.

[4] R. A. Howard. *Dynamic Programming and Markov Processes*. John Wiley, 1960. DOI: 10.2307/3611804.

[5] B. Bonet and H. Geffner. "Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming". In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. 2003, pp. 12–21.

[6] E. A. Hansen and S. Zilberstein. "LAO*: A heuristic search algorithm that finds solutions with loops". In: *Artificial Intelligence* 129.1-2 (2001), pp. 35–62. DOI: 10.1016/S0004-3702(01)00106-0.

[7] K. Goto and R. Van De Geijn. "Anatomy of high-performance matrix multiplication". In: *ACM Transactions on Mathematical Software* 34.3 (2008). DOI: 10.1145/1356052.1356053.

[8] A. LaMarca and R. E. Ladner. "The Influence of Caches on the Performance of Sorting". In: *Journal of Algorithms* 31.1 (1999), pp. 66–104. DOI: 10.1006/jagm.1998.0985.

[9] S. Han, L. Zou, and J. X. Yu. "Speeding up set intersections in graph algorithms using SIMD instructions". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 2018, pp. 1587–1602. ISBN: 9781450317436. DOI: 10.1145/3183713.3196924.


[2] https://cria2.uqam.ca/~jgareau/csr_mdp.zip

[10] G. Henry, P. T. P. Tang, and A. Heinecke. "Leveraging the bfloat16 Artificial Intelligence Datatype for Higher-Precision Computations". In: *Proceedings of the Symposium on Computer Arithmetic*. Institute of Electrical and Electronics Engineers Inc., 2019, pp. 69–76. ISBN: 9781728133669. DOI: 10.1109/ARITH.2019.00019.

[11] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell. "Bfloat16 Processing for Neural Networks". In: *Proceedings of the Symposium on Computer Arithmetic*. Institute of Electrical and Electronics Engineers Inc., 2019, pp. 88–91. ISBN: 9781728133669. DOI: 10.1109/ARITH.2019.00022.

[12] R. Raina, A. Madhavan, and A. Y. Ng. "Large-scale deep unsupervised learning using graphics processors". In: *Proceedings of the 26th International Conference On Machine Learning, ICML 2009*. 2009, pp. 873–880. ISBN: 9781605585161.

[13] A. Jain, A. A. Awan, A. M. Aljuhani, J. M. Hashmi, Q. G. Anthony, H. Subramoni, D. K. Panda, R. MacHiraju, and A. Parwani. "Gems: Gpu-enabled memory-aware model-parallelism system for distributed dnn training". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. ACM, 2020. ISBN: 9781728199986. DOI: 10.1109/SC41405.2020.00049.

[14] C. Guo, B. Y. Hsueh, J. Leng, Y. Qiu, Y. Guan, Z. Wang, X. Jia, X. Li, M. Guo, and Y. Zhu. "Accelerating sparse DNN models without hardware-support via tile-wise sparsity". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. ACM, 2020. ISBN: 9781728199986. DOI: 10.1109/SC41405.2020.00020.

[15] A. W. Moore and C. G. Atkeson. "Prioritized sweeping: Reinforcement learning with less data and less time". In: *Machine Learning* 13.1 (1993), pp. 103–130. DOI: 10.1007/BF00993104.

[16] D. Wingate and K. D. Seppi. "Prioritization methods for accelerating MDP solvers". In: *Journal of Machine Learning Research* 6 (2005), pp. 851–881.

[17] P. Dai, Mausam, D. Weld, and J. Goldsmith. "Topological value iteration algorithms". In: *Journal of Artificial Intelligence Research* 42 (2011), pp. 181–209. DOI: 10.1613/jair.3390.

[18] D. P. Bertsekas. *Dynamic programming and optimal control*. Vol. 2. Athena scientific Belmont, MA, 2001.

[19] H. B. McMahan, M. Likhachev, and G. J. Gordon. "Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees". In: *Proceedings of the 22nd International Conference on Machine Learning (ICML'05)*. 2005, pp. 569–576. ISBN: 1595931805. DOI: 10.1145/1102351.1102423.

[20] A. Jain and S. Sahni. "Cache efficient Value Iteration using clustering and annealing". In: *Computer Communications* 159 (2020), pp. 186–197. DOI: 10.1016/j.comcom.2020.04.058.

[21] D. Wingate and K. D. Seppi. "Cache performance of priority metrics for MDP solvers". In: *AAAI Workshop - Technical Report*. Vol. WS-04-08. AAAI Press, 2004, pp. 103–106.

[22] E. Bargiacchi, D. M. Roijers, and A. Nowé. "AI-Toolbox: A C++ library for Reinforcement Learning and Planning (with Python Bindings)". In: *Journal of Machine Learning Research* 21.102 (2020), pp. 1–12. URL: http://jmlr.org/papers/v21/18-402.html.

[23] A. Kolobov, Mausam, and D. S. Weld. "LRTDP versus UCT for online probabilistic planning". In: *Proceedings of the National Conference on Artificial Intelligence*. 2012, pp. 1786–1792. ISBN: 9781577355687. URL: https://ojs.aaai.org/index.php/AAAI/article/view/8362.

[24] B. Wheatman and H. Xu. "Packed Compressed Sparse Row: A Dynamic Graph Representation". In: *IEEE High Performance Extreme Computing Conference, HPEC 2018*. Institute of Electrical and Electronics Engineers Inc., 2018. ISBN: 9781538659892. DOI: 10.1109/HPEC.2018.8547566.

[25] C. Qian, B. Childers, L. Huang, H. Guo, and Z. Wang. "CGAcc: A compressed sparse row representation-based BFS graph traversal accelerator on hybrid memory cube". In: *Electronics (Switzerland)* 7.11 (2018). DOI: 10.3390/electronics7110307.

[26] B. Bonet and H. Geffner. "Learning depth-first search: A unified approach to heuristic search in deterministic and non-deterministic settings, and its application to MDPs". In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. 2006, pp. 142–151.