ECAI 2023
K. Gal et al. (Eds.)

© 2023 The Authors.
This article is published online with Open Access by IOS Press and distributed under the terms
of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/FAIA230293

Cache-Efficient Dynamic Programming MDP Solver

Jaël Champagne Gareau[a;*], **Guillaume Gosset**[a], **Éric Beaudry**[a] **and Vladimir Makarenkov**[a]

[a]Computer Science Department, Université du Québec à Montréal, Montréal, Québec, Canada
ORCiD ID: Jaël Champagne Gareau https://orcid.org/0000-0002-1906-4157,
Guillaume Gosset https://orcid.org/0000-0003-4898-0602, Éric Beaudry https://orcid.org/0000-0002-4460-0556,
Vladimir Makarenkov https://orcid.org/0000-0003-3753-5925

Abstract.

Automated planning research often focuses on developing new algorithms to improve the computational performance of planners, but effective implementation can also play a significant role. Hardware features such as memory hierarchy can yield substantial running time improvements when optimized. In this paper, we propose two state-reordering techniques for the Topological Value Iteration (TVI) algorithm. Our first technique organizes states in memory so that those belonging to the same Strongly Connected Component (SCC) are contiguous, while our second technique optimizes state value propagation by reordering states within each SCC. We analyze existing planning algorithms with respect to their cache efficiency and describe domain characteristics which can provide an advantage to each of them. Empirical results show that, in many instances, our new algorithms, called eTVI and eiTVI, run several times faster than traditional VI, TVI, LRTDP and ILAO* techniques.

1 Introduction

Markov Decision Processes (MDPs) serve as a powerful tool for navigating decision-making scenarios in uncertain conditions. These scenarios typically involve an agent that must accomplish a goal by choosing actions from a range of possibilities. The agent's decisions are guided by a probabilistic model defining potential outcomes of these actions, which can either be known a priori in the context of automated planning [18], or learned through real-world or simulated experiments in the context of (Model-Based) Reinforcement Learning (RL) [23]. Once an MDP model is established, the primary objective is to derive an optimal policy, i.e., a mapping that outlines the action to be taken in each state to maximize reward (or to minimize cost). Dynamic programming algorithms such as Value Iteration (VI) [1] and Policy Iteration [14] are commonly employed in automated planning to find such a policy. Planning algorithms for solving MDPs have evolved since then to become more efficient, incorporating ideas from heuristic search algorithms and trial-based (sampling) algorithms. Labeled Real-Time Dynamic-Programming (LRTDP) [3] and LAO* [12] are examples of more contemporary planning algorithms that merge these two methodologies.

A promising approach to enhance the efficiency of MDP solvers involves leveraging the advanced capabilities of contemporary CPUs, including cache memory and vector (Single Instruction Multiple Data, SIMD) instructions. We argue that modifying existing MDP solvers to harness these features could yield substantial performance boosts, mirroring those observed in High-Performance Computing (HPC) [9, 17, 11]. In recent times, Machine Learning (ML) algorithms have seen dramatic performance improvements (across multiple orders of magnitude) by taking into account low-level computer architecture components. For example, the use of specialized floating-point numbers and SIMD instructions with these number types have provided significant computational accelerations in numerous ML applications [13, 5]. Additionally, parallelism, achieved via CPU or GPU, and memory strategies, such as tiling, have enabled many ML algorithms to address larger classification problems [21, 10].

Given the impressive performance enhancements realized in ML, it is plausible to anticipate that the application of similar techniques to AI planning could result in MDP solvers which are able of handling efficiently larger real-world problems than is currently feasible. In this paper, we show that state-of-the-art MDP solvers can run orders of magnitude faster if they exploit the memory hierarchy of modern computers. This problem has been previously recognized as very relevant by the automated planning community:

> "We believe that more research on cache efficiency of MDP algorithms is desirable and could lead to substantial payoffs [. . .] one may gain huge speedups due to better cache performance of the algorithms." [18]

The remainder of this paper is organized as follows. Section 2 presents a quick survey of existing MDP solvers and of recent attempts to consider hardware features in the context of MDP planning. Section 3 formally defines MDPs and other concepts used in our study. Section 4 presents the two proposed methods that optimize the use of cache memory by reordering the states according to, respectively, the graph topology between the strongly connected components (SCC) and the graph topology within each of them. We present our empirical evaluation study in Section 5, where we analyze the cache performance of existing MDP algorithms and show that our novel methods are capable of outperforming them in almost all considered instances. Finally, we conclude in Section 6.

2 Related Work

A significant number of MDP solvers draw their foundation from the Value Iteration (VI) algorithm [1], or more specifically, from asynchronous variants of VI. In these asynchronous versions, the order in which MDP states are backed up is flexible and does not require a

* Corresponding Author. Email: champagne_gareau.jael@uqam.ca.

uniform consideration. This flexibility can be exploited by assigning a priority to each state and processing all states based on this priority. Prioritized Sweeping [20] is an example of an algorithm that employs this strategy. However, the overhead of maintaining the priority queue that dictates the order of the state backups often negates the potential acceleration. A solution to this issue consists in grouping the states into partitions and assign priorities to these partitions rather than to individual states. This approach is adopted by the General Prioritized Solvers (GPS) family of algorithms, which have demonstrated speed improvements of two orders of magnitude on numerous MDP domains compared to Prioritized Sweeping [26]. A limitation of GPS is the absence of a universal method for state partitioning, necessitating efficient partitioning based on specific features of the MDP domain being considered.

In a more recent development, we need to mention the Topological Value Iteration (TVI) algorithm [8], which uses a generic method for partitioning MDP states. TVI takes into account the graphical structure of the MDP (i.e., the structure of the graph resulting from the all-outcomes determinization of the MDP) and employs Kosaraju's algorithm [22] to identify its strongly connected components (SCCs). TVI then applies VI to each SCC in reverse topological order. Given that, by definition, there are no cycles between SCCs, this order is optimal and each SCC only needs to be considered once [2]. TVI can significantly outperform general MDP solvers, such as LRTDP [3], BRTDP [19] and ILAO* [12], particularly on domains with numerous SCCs. MDP domains with state variables that can only change monotonically (either increase or decrease) will have many SCCs, and thus are well-suited for TVI. For instance, board games, where the total number of pieces available to each player can never increase, as in the game of chess, have many SCCs. A drawback of TVI is that SCCs can sometimes be large (in the worst case, an MDP includes only one SCC containing every state), and thus solving these SCCs with VI can be time-consuming. To address this issue, the Focused Topological Value Iteration (FTVI) algorithm employs a heuristic search to rapidly identify sub-optimal actions that can be pruned from the MDP, potentially allowing for the discovery of more SCCs [8]. However, FTVI's performance is heavily reliant on the informativeness of the lower and upper-bound heuristics it requires.

Some studies have considered hardware features of modern computers. For example, the P3VI (Partitioned, Prioritized, Parallel Value Iterator) algorithm, which is similar to the GPS family of algorithms mentioned above, but where partitions are solved in parallel [25]. Another parallel algorithm, pcTVI (parallel-chained TVI) [7], is based on TVI instead of GPS. It finds independent partitions that can be solved in parallel and has the advantage that no inter-thread communication is needed, thus having minimal overhead.

Some researches tried to improve the cache performance of MDP planners. There are two different ways to do it: (1) by changing the data-structures used to store the MDP in memory, and (2) by changing the algorithms per se. To the best of our knowledge, the former way has only been considered in one study [6] that proposed to store MDP instances in a new data-structure, called CSR-MDP, inspired by the compressed-sparse-row representation of graphs. This CSR-MDP memory representation consists of five arrays (S, C, A, N, P), where S contains the states' actions ids; C contains the cost of each action; A contains the actions' effects ids; N and P contain respectively the effects' state transitions and probabilities. The latter way of improving the cache performance has also been proposed recently [16]. The algorithm, called Cache-Efficient with Clustering (CEC), subdivides the SCCs found by the FTVI algorithm into groups of states of size that fits the L3 CPU cache memory. The step of FTVI consisting in solving an SCC using VI is replaced by a procedure that cyclically solves every cluster in the SCC until the entire SCC converges. The authors of CEC indicated that their algorithm allowed them to achieve a speedup factor varying between 2 and 8, compared to FTVI. Other works have considered cache memory of hard drives when MDP instances do not fit totally in the main memory [24], but we don't discuss them here since this problem is out of our scope.

3 Problem Definition

There exist different types of MDPs, including Finite-Horizon MDP, Infinite-Horizon MDP and Stochastic Shortest Path MDP (SSP-MDP) [18], where the first two of them can be seen as special cases of the last one of them [2]. Due to their greater generality, we therefore focus on SSP-MDPs in this work, which we describe formally in Definition 1 below. We refer less familiar readers to Mausam and Kolobov's book to gain background necessary to reproduce the results of our study [18].

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;
- $T\colon S \times A \times S \to [0, 1]$ gives the probability $T(s, a, s')$ 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 (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 π until a goal is reached has a minimal expected cost. The expected cost of following a policy π 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. For all $s \in S$,

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

The part between square brackets is called the *Q-value* of a state-action pair. When an optimal value function $V^\star$ (or the optimal Q-value function $Q^\star$) is known, an optimal policy $\pi^\star$ can be found greedily: $\pi^\star(s) = \operatorname{argmin}_{a \in A} Q^\star(s, a)$.

Many MDP solvers use dynamic programming algorithms like VI, which update iteratively an arbitrarily initialized value function until convergence with a given precision ϵ. In general SSP-MDP problems, VI takes a polynomial time on the number of states (it is P-Complete [18]). In acyclic MDPs, VI needs to do at most $|S|$ sweeps of the state space, where one sweep consists in updating the value estimate of every state using the Bellman Optimality Equations. Hence, in such MDPs, the number of state updates (called a *backup*) is $\mathcal{O}(|S|^2)$. However, in these acyclic MDPs, most of these backups are wasteful, since the MDP can in this situation be solved using only $|S|$ backups [2] (ordered in reverse topological order). Therefore, the state backup order can have a significant impact on the algorithm performance.

4 Cache-Efficient Topological Value Iteration

Modern computers have multiple types of memory, forming what is known as a memory hierarchy. From slowest (and largest) to fastest (and smallest), there is the data drive (hard drive or SSDs), the main (RAM) memory, the L3, L2, and L1 CPU cache memory, and the registers. The caches allow the CPU to retain recently used data for later use without necessity to access the much slower main memory. On modern CPUs, the smallest amount of data loaded at a time in memory, named the *cache line* size, is generally 64 bytes. The access time to the fastest level of cache is usually around three orders of magnitude faster than the access time to main memory. Moreover, the cost of an L3 cache-miss (i.e., a load of a memory address not currently in the L3 cache) on modern computers is two to three orders of magnitude greater than that of an arithmetic operation. For example, on a 2015 Skylake-based 6th generation Intel Core CPU, a cache-miss leads to a penalty of 50-70 cycles [15].

In this paper, we focus mostly on minimizing the amount of L3 cache misses. However, the proposed methods will also provide improved performance even if the problem fits entirely in this cache level. Here are two reasons for this: First, if an MDP fits entirely in a certain cache level, then the consideration of memory locality will help reduce the number of cache misses on the lower levels, and thus the provided advantages of the proposed methods will be similar; Second, by improving the locality of memory, we are also minimizing the amount of wasted memory inside each cache line (and therefore, wasted memory bandwidth).

In this section, we describe two cache-efficient versions of the TVI algorithm, named eTVI and eiTVI. We start by summarizing how the original TVI algorithm works. Algorithm 1 presents its main steps. First, TVI uses Kosaraju's graph algorithm on a given MDP to find the strongly connected components (SCCs) of its graphical structure (the graph corresponding to its all-outcomes determinization). The SCCs are found by Kosaraju's algorithm in reverse topological order, which means that for every $i < j$, there is no path from a state in the ith SCC to a state in the jth SCC. This property ensures that every SCC can be solved separately by VI sweeps if previous SCCs (according to the reverse topological order) have already been solved. The second step of TVI is thus to solve every SCC one by one in that order.

Algorithm 1 Topological Value Iteration

```
1: procedure TVI(M: MDP)
2:     ▷ SCCs are found in reverse topological order
3:     SCCs ← KOSARAJU(M)                 ▷ Or Tarjan's algorithm
4:     for all scc ∈ SCCs do
5:         PARTIALVI(M, scc)              ▷ converge with a given ϵ
```

Since TVI divides the MDP in multiple subparts, it maximizes the effectiveness of every state backup by ensuring that only converged state values are propagated from one SCC to the other. Another, perhaps less obvious, advantage of TVI is that since fewer states are considered in an SCC sweep (compared to a total state space sweep), the probability of having enough space in CPU cache memory to store the information required by the sweep is higher. Even when an SCC is too large to fit entirely in the cache memory, the number of cache-misses will be reduced compared to that obtained by doing a sweep over the entire MDP. Finally, depending on how TVI is implemented, there can also be a third factor explaining its improved performance. Let's assume that an MDP contains only one SCC. Will TVI behave similarly to VI in such a case? Not necessarily, because Kosaraju's algorithm (or Tarjan's algorithm) will find and store the (unique) SCC in a postorder depth-first search (DFS). If the PARTIALVI does the sweep using this order, it will most likely improve the information flow, and therefore be more efficient than the arbitrary order used by asynchronous VI.

Although the cache performance of TVI is better than that of VI, this is mostly "accidental" since this was not a direct motivation of TVI's authors. One way we can further improve the cache performance is to subdivide every SCC into smaller subsets of states whose information (transitions, state values, etc.) fit in the (e.g., L3) cache. This is the strategy used by the CEC algorithm mentioned in Section 2. Another way, on which we will focus next, is to reorder the data in the data structure(s) containing the MDP such that SCCs data are packed contiguously in memory, ensuring easily predictable memory access patterns and maximizing useful data content in loaded cache lines. For example, if an MDP has two SCCs containing respectively the states $(0, 2, 4, 6)$ and the states $(1, 3, 5, 7)$, we can reorder them so that the MDP states' data are stored in the order $(0, 2, 4, 6, 1, 3, 5, 7)$ instead of the original order $(0, 1, 2, 3, 4, 5, 6, 7)$. This example is not even the worst case: for instance, if we assume that each state of a four-state SCC occupies 16 contiguous bytes in memory and is stored on a different cache than the three other states, then solving the SCC would require loading four lines from RAM through all cache levels, whereas by making the states contiguous, only one cache line load (of 64 bytes) would be necessary.

The eTVI algorithm we propose (Algorithm 2) here uses this reordering idea to reduce the amount of memory accesses and thus improve the computation speed. However, reordering the states in memory to improve cache performance does not make much sense if the MDP is stored in a cache-inefficient data-structure (e.g., using linked-lists). Therefore, our algorithm assumes that the MDP is stored using the CSR-MDP memory representation mentioned in Section 2.

It is worth noting that the proposed eTVI algorithm is based on TVI, but it can also directly be incorporated into FTVI. However, the additional step carried out by FTVI, i.e., pruning suboptimal actions, is done by a heuristic search algorithm. To the best of our knowledge, nobody has studied the cache performance of such algorithms (e.g., LRTDP and ILAO*). Consequently, we decided not to use FTVI in our implementation of eTVI to focus specifically on its impact on cache performance.

The eTVI algorithm first determines the k SCCs as in TVI, but uses Tarjan's algorithm instead of Kosaraju's algorithm since it is about twice faster. It then assigns a new id to every state in the MDP, such that there exist ids $i_0 < i_1 < \cdots < i_k$ (given by the array $SCCsIDs$ in the pseudocode) that ensure that all states in the jth SCC have an id i, where $i_j \leq i < i_{j+1}$. Once a new id has been assigned to every state, we iterate over each state, action and probabilistic action effect to rebuild a CSR-MDP memory representation of the MDP using the new indices. After this step, the five arrays of the CSR-MDP representation will be implicitly divided into k contiguous regions such that the ith region of an array contains only data specific to the ith SCC. It is worth noting that we could have generated a different 5-arrays tuple for every SCC, but we preferred to implicitly divide the same five arrays into different regions to improve memory locality. When the reordering/rebuilding of the MDP is done, we solve each SCC one-by-one in reverse topological order (as in TVI) by considering regions of the five arrays one-by-one. Since SCCs are stored contiguously in the five arrays, the number of wasted bytes in every cache line stored in cache memory is minimal.

Algorithm 2 Cache-Efficient Topological Value Iteration

```
1: procedure ETVI(M: MDP (stored using CSR-MDP))
2:     ▷ SCCs are found in reverse topological order
3:     SCCs ← TARJAN(M)                          ▷ array of arrays
4:     SCCsIDs ← REORDER(M, SCCs)
5:     for i ← 0 to SIZE(SCCs) − 1 do
6:         startID ← SCCsIDs[i]
7:         endID ← SCCsIDs[i + 1]                ▷ last is excluded
8:         PARTIALVI(M, startID, endID)
9: procedure REORDER(M, SCCs)
10:    ▷ Step 1: Assign a new id to every state
11:    (S, C, A, N, P) ← M                       ▷ Unpack CSR-MDP arrays
12:    n ← NUMSTATES(M)
13:    k ← SIZE(SCCs)                            ▷ number of SCCs
14:    SCCsIDs ← array of capacity k + 1
15:    INSERT(SCCsIDs, 0)
16:    oldIDs, newIDs ← arrays of size n
17:    current ← 0
18:    for all scc ∈ SCCs do
19:        for all stateID ∈ scc do
20:            oldIDs[current] = stateID
21:            newIDs[stateID] = current
22:            current ← current + 1
23:        INSERT(SCCsIDs, current)
24:    REBUILDCSR(M, oldIDs, newIDs)
25:    return SCCsIDs
26: procedure REBUILDCSR(M, oldIDs, newIDs)
27:    (S', C', A', N', P') ← new arrays of the same size
28:    for s_id^old ← 0 to n − 1 do              ▷ over all states
29:        for all a_id^old ∈ S_s do             ▷ over all actions of s
30:            for all e_id^old ∈ A_a do         ▷ over all effects of a
31:                update N' and P'
32:            update A' and C'
33:        update S'
34:    M ← (S', C', A', N', P')
```

So far, we have only considered the order of states in memory with respect to the SCC they're part of, which we call the order of states *extra-SCC*. What about the order of states *within* an SCC (which we call the order of states *intra-SCC*)? As mentioned previously, TVI (and for the same reason, eTVI) can sometimes be faster than VI even on MDPs having only one SCC, because it does the VI sweep using a different states order. This order will be the one found by the SCC algorithm (e.g., a postorder DFS order). Is it possible to find a better one? Since, by definition, an SCC contains loops, the optimal order may change after each sweep. Algorithms that find a dynamic state ordering, such as Prioritized Value Iteration [26], have a large overhead because they need to maintain a priority queue that returns the states in the dynamic order. Instead, we propose a static backup order for each SCC given by a reversed Breadth-First Search (BFS) where, at the start, the BFS queue contains all outward SCC border states (that we define in Definition 3).

Definition 3. Let $M = (S, A, T, C, G)$ be an MDP and K be an SCC of M. We say that an *outward border state* of K is a state $s \in K$ for which there exists an action $a \in A$ and a state $s' \in S \setminus K$ such that $T(s, a, s') > 0$.

By starting the reversed BFS at the outward border states of the current SCC, we make sure that the first backups within the SCC will "bring" into the SCC the already converged values of the outward neighbor SCC. After that, the reversed BFS order ensures that the state-values will be propagated so that each state backup has new values of neighboring states to consider (i.e., no state backup will ever be useless). Since the SCC containing the goal state(s) do(es) not have any outward border state, we simply start the reversed BFS from the goal state. The algorithm that combines eTVI with this intra-SCC order is called eiTVI. The eiTVI algorithm is implemented by replacing lines 19–22 in Algorithm 2 by a detection of the outward border states and the reversed BFS from those states. The names eTVI and eiTVI stand respectively for *extra-TVI* and *extra-intra-TVI*, since the former reorders the states according to an extra-SCC order, whereas the latter reorders the states according to both the extra-SCC and intra-SCC orders.

5 Empirical Evaluation

5.1 Evaluated Planners

In this section, we compare the performance of our proposed algorithms, eTVI and eiTVI, to the following traditional algorithms: (1) VI (we use the asynchronous round-robin Gauss-Seidel variant), (2) LRTDP, (3) ILAO*, and (4) TVI – the Topological Value Iteration algorithm described in Section 4. For LRTDP and ILAO*, we used the admissible and domain-independent $h_{\min}$ heuristic first described in the original paper introducing LRTDP [3]:

$$h(s) = \begin{cases} 0 & \text{if } s \in G, \\ \min_{a \in A_s} \left[C(s,a) + \min_{s' \in S} T(s,a,s') h_{\min}(s') \right] & \text{otherwise,} \end{cases}$$

where A_s denotes the set of applicable actions in state s. We computed the h_{min} heuristic using the same method proposed in TVI's original paper, i.e., we do a backward search in the graph corresponding to the all-outcomes determinization.

The six competing algorithms (VI, LRTDP, ILAO*, TVI, eTVI and eiTVI) were implemented in C++ by the authors of this paper, and compiled using the GNU g++ compiler (version 12.2). Our TVI implementation (as well as eTVI and eiTVI) uses Tarjan's algorithm instead of Kosaraju's algorithm to compute the SCCs. All tested MDPs were stored with the CSR-MDP memory representation. All our experiments were performed on a computer equipped with a 4.2 GHz Intel Core i5-7600k processor and 16 GB of DDR4 RAM.

Unfortunately, we were unable to reproduce the results reported in CEC[A]'s research paper [16] (we obtained a significantly smaller speedup factor than reported in the paper). Therefore, we do not include these methods (discussed in Section 2) in our benchmark. When implementing these algorithms, we found that the computational overhead required to find the "external states" and the SCC subsets fitting in L3 cache outshone the obtained marginal cache improvement. We suspect that the difference between what we observed and what the authors of the CEC[A] algorithms claim is due to the poor cache performance of their MDP memory representation, which bias their results. Indeed, all of their compared algorithms (including their baseline, FTVI) are implemented with MDPs stored using a linked-list of linked-lists, which leads to many (unrealistic) cache-improvement opportunities that we cannot reproduce when the MDP is stored in a cache-efficient way, as is the case of CSR-MDP. Therefore, the implementation used by CEC[A]'s authors largely overestimates the potential cache improvement of their methods (e.g., our simple VI implementation was faster than their C++ implementation of CEC[A]).

5.2 Domains

Note that TVI (and therefore, also eTVI and eiTVI) are designed for solving enumerative SSP-MDPs, i.e., the problems described explicitly, in contrast to the problems described in a factored form (e.g., in PPDDL or RDDL). Therefore, we evaluated the performance of the algorithms on three different enumerative MDP domains.

The first domain is the generic Layered domain described in TVI's paper [8]. 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 $\{i, i+1, \ldots, n_l\}$, which means that MDPs in this layered domain have at least n_l SCCs. We used $n_a = 10$ and $n_s = 10$ in each test instance.

The second domain considered is the Single-Armed Pendulum (SAP) domain [26]. 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 θ and angular velocity ω. We used the SAP instance generator proposed by David Wingate. The details of the discretisation of the state space are explained in his paper [26].

Finally, the third domain is a variant of the Wetfloor domain [4]. In this domain, the state space is a square navigation grid in which cells can be either dry, slightly wet or heavily wet. In the grid, cells are independently assigned 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}. We fixed the parameters to $p = 0.4$; $q = 0.5$; $r_{slightly} = 0.2$; and $r_{heavy} = 0.33$. 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). This variant allows us to measure what happens to the cache performance of the tested algorithms when the number of SCCs is increased.

We also evaluated the performance of the compared algorithms on the DAP (Double-Armed Pendulum) and the MCar (Mountain Car) domains, but the results obtained for these domains were similar to those obtained on the three mentioned domains, and therefore we do not include them due to lack of space.

5.3 Results

For every test domain, we measured the running time of the compared algorithms carried out until ϵ-convergence of the value function (ϵ was fixed to 10^{-6} in our study). For every tested parameter configurations, we ran each competing algorithm on the same 15 randomly generated MDP instances. Figure 1 illustrates the results of the six competing algorithms on the tested domains. For the Layered domain, we present two figures: one where the number of layers is fixed (to 10) and the number of states varies (from 100k to 1M), and one where the number of states is fixed (to 1M) and the number of layers

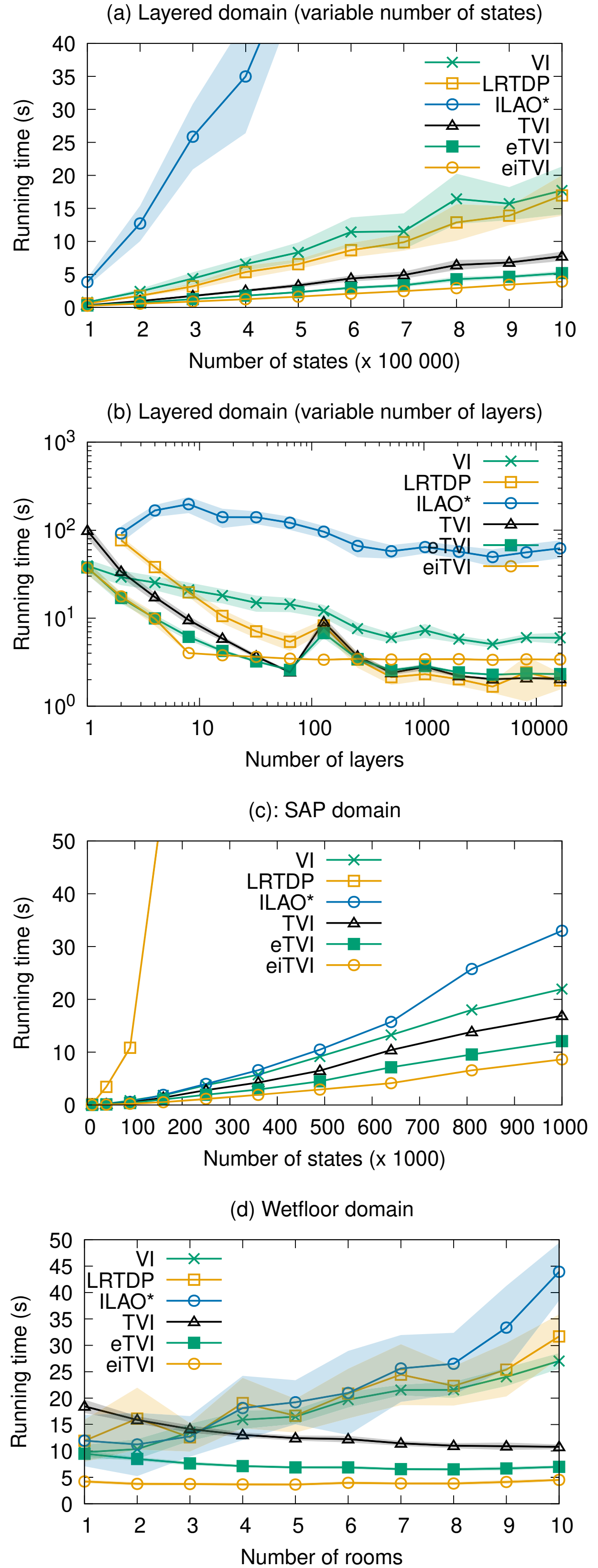

Figure 1. Average running times (in s) and 95% confidence intervals of the six competing algorithms for: (a) the Layered domain with varying number of states and fixed number of layers (10), (b) the Layered domain with varying number of layers and fixed number of states (1M), (c) the SAP domain, and (d) the Wetfloor domain.

varies (from 2^0 to 2^{14}). On each graph plot, we also represented the 95% confidence interval for each algorithm. Since the SAP domain generator is deterministic, the 15 instances for each tested size are, in fact, the same for this domain. The very small variations observed in this case are due to environmental biases (e.g., background tasks running on the machine), and not to the differences between the generated instances.

Table 1 reports the detailed results provided for different instances of the considered Layered, SAP and Wetfloor domains by the TVI, eTVI and eiTVI algorithms[1]. The first four columns account for the characteristics of the considered MDP instances, including the domain name, the number of states of the generated instance, the number of SCCs (we exclude the SCCs containing only one state) and the size of the largest SCC. The next column accounts for the running time of Tarjan's algorithm used to find the SCCs. Finally, we report for TVI, eTVI and eiTVI the time required to compute the state reordering and to build the new CSR-MDP memory representation (T_r), the time required to solve sequentially each SCC in the reversed topological order (T_s), the total running time of the algorithm (T_{tot}) and, finally, the number of Bellman backups (B) that were necessary to reach epsilon-convergence of the value-function.

At the beginning of our empirical evaluation, we measured directly the number of L3 cache-references and cache-misses[2]. However, when we get to this level of measurement, many factors, including the hardware prefetcher or the L1 and L2 caches, can have an important bias on the L3 cache metrics being measured. Therefore, we would need to add multiple new metrics to our study. Since we were limited by the paper length, we decided to present our results using the number of Bellman backups, which might be of greater interest to the AI community and allows us to measure indirectly the cache performance. This way, we can use two criteria (running time and number of Bellman backups) to compare the competing methods. For example, if we compare the results of TVI and eTVI reported in column (B) of Table 1, we can see that they are identical (this is expected, since the only change between TVI and eTVI concerns the way the states are stored in memory, but not their sweeping order). Since the number of backups in the two algorithms is the same, but the running time of eTVI is lower, this supports our hypothesis that the reordering of the states leads to a better cache performance of eTVI. We can also use the number of Bellman backups to show that eiTVI focuses on an improved state-value propagation order in addition to optimizing the cache-performance as it has a lower number of backups than TVI and eTVI.

In the Layered domain, we can see that the TVI, eTVI and eiTVI algorithms greatly outperformed VI, LRTDP and ILAO*. Moreover, we can observe that eTVI and eiTVI's advantage over TVI seems to increase as the number of states increases. When varying the number of layers, we can see that eTVI and eiTVI are faster than the other algorithms up until a certain point (in our case, 128 layers), after which the number of layers becomes so high that the number of states per layer becomes small enough to fit in cache. When this happens, cache-misses will only occur on the first sweep in the SCC. The states' reordering will minimize this (small) number of cache-misses but, in this situation (small SCCs), TVI is already very fast, and the performance increase by the smaller number of cache-misses is negligible compared to the cost of the reordering (this cost is around 1% of the total execution time when SCCs are large, but can be as much as 30% of the total time when the solving of SCCs is already extremely fast as is the case of small SCCs). LRTDP is also affected by the number of layers, even though it does not explicitly consider SCCs, because the number of layers has an impact on the search depth attained by LRTDP before it reaches a goal.

On the SAP domain, LRTDP has a lot of difficulty to find a solution in a reasonable time (TVI's authors have observed the same results). This can be explained by the fact that the $h_{\min}$ heuristic is not really informative in SAP as well as by the fact that the minimum number of actions needed to reach a goal is relatively high. Interestingly, the difference between eiTVI and eTVI is similar to the difference between eTVI and TVI, which is also similar to the difference between TVI and VI. This finding confirms that eTVI and eiTVI can provide important speedup even for domains containing a single SCC. In the case of eTVI, the speedup in single SCC domains, such as SAP, is due to the memory organisation differences. In fact, even though TVI and eTVI use the same state backup order, the order in which the states are stored in memory when using TVI does not necessarily match the order in which the states are considered in successive Bellman backups. In contrast, eTVI rebuilds the MDP in memory in a way that these two orders match, which improves the cache-performance. In the case of eiTVI, the speedup in single SCC domains is due to the fact that eiTVI reorders the states in the order given by a reversed BFS from the goal state. Therefore, the state-value propagation order is improved.

On the Wetfloor domain, we can see unsurprisingly that both TVI and eTVI's performance increase as the number of rooms increases. More surprising is the fact that eiTVI seems pretty constant even as the number of rooms change. This is because the information flow with eiTVI is initially much better and therefore, adding new SCCs leads to minimal opportunities of additional improvement. For example, when comparing the (B) column of eiTVI and eTVI, we see that eiTVI needs less than half of the Bellman backups before ϵ-convergence. In comparison, VI's performance decreases as the amount of room increases. This is because 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 and ILAO*, the poor performance when the number of room gets higher can be explained by the fact that a higher number of rooms corresponds to a larger search depth from the initial state to the goal state in the Wetfloor domain, and a less informative heuristic function.

We also evaluated the performance of the compared algorithms on an Intel Core i5-13500 CPU with 24MB of L3 cache [3] to assess the impact of newer CPU cache architecture. For the SAP and Layered domains, each of the competing algorithms was about 2-3 times faster on the i5-13500 CPU, but the gap between the algorithms was almost the same. For the Wetfloor domain, the gap between TVI and eTVI was about 50% smaller on the i5-13500 CPU than on the i5-7600k CPU, but the gap between eTVI and eiTVI was similar.

Overall, the proposed eTVI and eiTVI algorithms clearly outperform their VI, TVI, ILAO* and LRTDP counterparts on almost every MDP instance (see Table 1). They are particularly good when an MDP domain has many large SCCs, but can also outperform the other algorithms when there is only one SCC (e.g., the SAP domain). The only case where TVI was faster than both eTVI and eiTVI was when the number of SCCs was large, but each of them was small.

[1] A more detailed version of this table reporting the 95% confidence intervals and a second table containing the data for the six compared algorithms (the data used to plot the figures) are available in Supplementary Material. We also included in it two tables showing, respectively, the obtained cache metrics (cache references and cache misses) and the average speedup factors obtained for each tested domains. It can be downloaded from: https://www.jaelgareau.com/en/publication/gareau-ecai23/supp.pdf.

[2] By using the Linux `perf` command.

[3] Intel released this CPU in 2023Q1, we received it during last revision of this paper.

Table 1. Average running times (in s) and the number of Bellman backups for every tested domain and each of the TVI, eTVI and eiTVI algorithms. Fastest total time (T_{tot}) for each domain instance is bolded.

MDP instances characteristics				Tarjan	TVI			eTVI				eiTVI			
D	$\lvert S\rvert$ (k)	$\lvert K\rvert$	$\lvert k_{\max}\rvert$ (k)		T_s	T_{tot}	B (M)	T_r	T_s	T_{tot}	B (M)	T_r	T_s	T_{tot}	B (M)
Layered	100	10	10	0.049	0.276	0.328	1.43	0.052	0.194	0.298	1.43	0.125	0.077	**0.253**	0.552
	200	10	20	0.115	0.844	0.964	3.53	0.111	0.509	0.74	3.53	0.266	0.172	**0.561**	1.14
	300	10	30	0.196	1.53	1.74	5.88	0.176	0.87	1.25	5.88	0.410	0.273	**0.886**	1.73
	400	10	40	0.288	2.24	2.54	8.18	0.246	1.23	1.78	8.18	0.559	0.376	**1.24**	2.32
	500	10	50	0.391	2.92	3.33	10.4	0.320	1.59	2.31	10.4	0.739	0.479	**1.62**	2.89
	600	10	60	0.509	3.85	4.38	13.3	0.399	2.04	2.97	13.3	0.945	0.592	**2.08**	3.52
	700	10	70	0.632	4.25	4.89	14.1	0.483	2.22	3.36	14.1	1.129	0.689	**2.47**	4.09
	800	10	80	0.760	5.64	6.42	18.4	0.562	2.92	4.27	18.4	1.335	0.804	**2.92**	4.68
	900	10	90	0.887	5.89	6.8	18.6	0.652	3.06	4.63	18.6	1.602	0.923	**3.44**	5.29
	1000	10	100	1.026	6.69	7.74	20.4	0.736	3.41	5.19	20.4	1.811	1.05	**3.9**	5.89
Layered	1000	1	1000	1.309	97.5	98.8	197	0.919	35.5	37.7	197	1.537	34.5	**37.3**	191
	1000	2	500	1.332	32.5	33.9	83.2	0.853	14.7	**17**	83.2	1.759	14.6	17.7	82
	1000	4	250	1.201	16.1	17.4	46.2	0.786	7.93	9.94	46.2	1.819	6.81	**9.86**	39.3
	1000	8	125	1.062	8.46	9.54	25.8	0.750	4.3	6.13	25.8	1.824	1.11	**4.01**	6.16
	1000	16	62.5	0.955	4.89	5.87	15.9	0.716	2.54	4.24	15.9	1.835	0.965	**3.79**	5.5
	1000	32	31.3	0.899	2.7	3.67	9.52	0.691	1.56	**3.22**	9.52	1.906	0.756	3.64	4.17
	1000	64	15.63	0.836	1.4	**2.44**	5.06	0.687	0.875	2.59	5.06	1.963	0.485	3.47	2.3
	1000	128	7.81	0.773	8.04	9.04	31.1	0.675	5.11	6.77	31.1	1.822	0.58	**3.37**	2.81
	1000	256	3.91	0.831	2.57	3.69	10	0.670	1.67	**3.43**	10	1.955	0.37	3.45	1.44
	1000	512	1.96	0.821	1.25	**2.39**	4.33	0.680	0.763	2.57	4.33	2.015	0.254	3.39	0.782
	1000	1024	0.98	0.837	1.64	**2.8**	6.13	0.691	1.02	2.85	6.13	2.001	0.286	3.42	0.878
	1000	2048	0.49	0.823	1.05	**2.21**	3.37	0.684	0.606	2.42	3.37	2.048	0.233	3.42	0.602
	1000	4096	0.25	0.818	0.867	**2.02**	2.48	0.682	0.476	2.28	2.48	2.025	0.211	3.36	0.54
	1000	8192	0.12	0.826	0.92	**2.09**	2.77	0.686	0.529	2.35	2.77	2.045	0.207	3.41	0.474
	1000	16384	0.06	0.824	0.879	**2.04**	2.57	0.684	0.497	2.32	2.57	2.037	0.204	3.39	0.447
SAP	10	1	10	0.001	0.011	0.012	1.04	0.001	0.01	0.011	1.04	0.001	0.008	**0.009**	0.81
	40	1	40	0.002	0.102	0.105	9.08	0.002	0.092	0.098	9.08	0.005	0.057	**0.065**	5.6
	90	1	90	0.004	0.431	0.438	32.2	0.005	0.363	0.375	32.2	0.012	0.211	**0.23**	18.6
	160	1	160	0.008	1.37	1.39	88.2	0.009	1.01	1.04	88.2	0.022	0.491	**0.526**	42.7
	250	1	250	0.013	2.79	2.81	165	0.015	1.9	1.94	165	0.036	1.1	**1.15**	95.2
	360	1	360	0.018	4.21	4.24	250	0.021	2.87	2.92	250	0.054	1.83	**1.92**	160
	490	1	490	0.027	6.46	6.5	377	0.029	4.41	4.48	377	0.075	2.8	**2.92**	241
	640	1	640	0.037	10.3	10.4	600	0.038	7.04	7.14	600	0.100	3.97	**4.12**	339
	810	1	810	0.045	13.8	13.8	813	0.050	9.44	9.56	813	0.130	6.37	**6.56**	549
	1000	1	1000	0.055	16.8	16.9	1020	0.061	11.9	12.1	1020	0.162	8.39	**8.63**	718
Wetfloor	500	1	500	0.053	18.4	18.4	413	0.054	9.36	9.48	413	0.104	4.04	**4.22**	180
	500	2	250	0.051	15.8	15.9	372	0.053	8.37	8.49	372	0.097	3.6	**3.76**	161
	500	3	166	0.050	14.1	14.1	336	0.053	7.52	7.64	336	0.095	3.58	**3.74**	160
	500	4	125	0.049	13	13	314	0.053	7.02	7.13	314	0.097	3.51	**3.67**	157
	500	5	100	0.048	12.4	12.4	303	0.053	6.77	6.88	303	0.093	3.49	**3.64**	157
	500	6	83.5	0.048	12.2	12.2	304	0.053	6.77	6.88	304	0.094	3.81	**3.96**	171
	500	7	71.3	0.046	11.4	11.4	289	0.052	6.44	6.55	289	0.093	3.69	**3.84**	166
	500	8	62.5	0.046	10.9	11	289	0.053	6.4	6.52	289	0.092	3.68	**3.83**	166
	500	9	55.7	0.046	10.8	10.9	299	0.052	6.57	6.68	299	0.092	3.97	**4.13**	181
	500	10	50.2	0.046	10.7	10.8	316	0.052	6.88	6.99	316	0.093	4.36	**4.52**	201

6 Conclusion

The main contributions of this paper are two-fold. First, we proposed two cache-efficient variants of the TVI algorithm, called eTVI and eiTVI, that reorder the original states of an MDP according to the SCC order (and, in the case of eiTVI, according to the order of states within SCCs as well) and build a corresponding reordered CSR-MDP representation such that the states frequently used together in computations are located near each other in memory. Second, we evaluated the cache performance of six competing algorithms (VI, LRTDP, ILAO*, TVI, eTVI and eiTVI) on three planning domains and analyzed the main factors (general, or specific to some solvers) that may impact their cache performance. Our C++ implementation, including domains generators, benchmarking scripts, and the proposed eTVI and eiTVI algorithms, are available online[4].

In the future, we plan to design and evaluate an algorithm that subdivides each SCC into smaller subcomponents with minimal dependencies between them, possibly by finding provably suboptimal strong-bridges within the SCCs and by using the Louvain algorithm. We also plan to work on improving the cache efficiency of heuristic search algorithms. This problem is harder to tackle than that of improving VI/TVI-based methods because the states visiting order is more difficult to predict. This would allow algorithms such as LRTDP and LAO*, but also FTVI, to leverage cache memory hierarchy as it was done for TVI in this paper.

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).

[4] https://www.jaelgareau.com/en/publication/gareau-ecai23/code.zip.

References

[1] Richard Bellman, *Dynamic Programming*, Prentice Hall, 1957.

[2] Dimitri P Bertsekas, *Dynamic programming and optimal control*, volume 2, Athena scientific Belmont, MA, 2001.

[3] Blai Bonet and Héctor Geffner, 'Labeled RTDP: Improving the convergence of real-time dynamic programming', in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 12–21, (2003).

[4] Blai Bonet and Héctor 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)*, pp. 142–151, (2006).

[5] Neil Burgess, Jelena Milanovic, Nigel Stephens, Konstantinos Monachopoulos, and David Mansell, 'Bfloat16 processing for neural networks', in *Proceedings of the Symposium on Computer Arithmetic*, pp. 88–91. Institute of Electrical and Electronics Engineers Inc., (2019).

[6] Jaël Champagne Gareau, Éric Beaudry, and Vladimir Makarenkov, 'Cache-efficient memory representation of Markov decision processes', *Proceedings of the Canadian Conference on Artificial Intelligence*, (2022).

[7] Jaël Champagne Gareau, Éric Beaudry, and Vladimir Makarenkov, 'pcTVI: Parallel MDP solver using a decomposition into independent chains', in *Classification and Data Science in the Digital Age – IFCS 2022*, Studies in Classification, Data Analysis, and Knowledge Organization, Cham, (2023). Springer International Publishing.

[8] Peng Dai, Mausam, Daniel Weld, and Judy Goldsmith, 'Topological value iteration algorithms', *Journal of Artificial Intelligence Research*, **42**, 181–209, (2011).

[9] Kazushige Goto and Robert Van De Geijn, 'Anatomy of high-performance matrix multiplication', *ACM Transactions on Mathematical Software*, **34**(3), (2008).

[10] Cong Guo, Bo Yang Hsueh, Jingwen Leng, Yuxian Qiu, Yue Guan, Zehuan Wang, Xiaoying Jia, Xipeng Li, Minyi Guo, and Yuhao 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).

[11] Shuo Han, Lei Zou, and Jeffery Xu Yu, 'Speeding up set intersections in graph algorithms using SIMD instructions', in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1587–1602. Association for Computing Machinery, (2018).

[12] Eric A Hansen and Shlomo Zilberstein, 'LAO*: A heuristic search algorithm that finds solutions with loops', *Artificial Intelligence*, **129**(1-2), 35–62, (2001).

[13] Greg Henry, Ping Tak Peter Tang, and Alexander Heinecke, 'Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations', in *Proceedings of the Symposium on Computer Arithmetic*, pp. 69–76. Institute of Electrical and Electronics Engineers Inc., (2019).

[14] R. A. Howard, *Dynamic Programming and Markov Processes*, John Wiley, 1960.

[15] Intel, *Intel® 64 and IA-32 architectures optimization reference manual*, 2022.

[16] Anuj Jain and Sartaj Sahni, 'Cache efficient value iteration using clustering and annealing', *Computer Communications*, **159**, 186–197, (2020).

[17] Anthony LaMarca and Richard E Ladner, 'The influence of caches on the performance of sorting', *Journal of Algorithms*, **31**(1), 66–104, (1999).

[18] Mausam and Andrey Kolobov, *Planning with Markov Decision Processes: An AI Perspective*, number 1, Morgan & Claypool, 2012.

[19] H. Brendan McMahan, Maxim Likhachev, and Geoffrey 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)*, pp. 569–576, (2005).

[20] Andrew W Moore and Christopher G Atkeson, 'Prioritized sweeping: Reinforcement learning with less data and less time', *Machine Learning*, **13**(1), 103–130, (1993).

[21] Rajat Raina, Anand Madhavan, and Andrew Y Ng, 'Large-scale deep unsupervised learning using graphics processors', in *Proceedings of the 26th International Conference On Machine Learning, ICML 2009*, pp. 873–880, (2009).

[22] Sharir, M, 'A strong-connectivity algorithm and its applications in data flow analysis', *Computers & Mathematics with Applications*, **7**(1), 67–72, (1981).

[23] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning, Second Edition: An Introduction*, MIT Press, 2018.

[24] David Wingate and Kevin D Seppi, 'Cache performance of priority metrics for MDP solvers', in *AAAI Workshop - Technical Report*, volume WS-04-08, pp. 103–106. AAAI Press, (2004).

[25] David Wingate and Kevin D Seppi, 'P3VI: A partitioned, prioritized, parallel value iterator', in *Proceedings of the Twenty-First International Conference on Machine Learning, ICML 2004*, pp. 863–870, (2004).

[26] David Wingate and Kevin D Seppi, 'Prioritization methods for accelerating MDP solvers', *Journal of Machine Learning Research*, **6**, 851–881, (2005).