Computer Communications 159 (2020) 186–197

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Cache efficient Value Iteration using clustering and annealing ☆

Anuj Jain [a,*], Sartaj Sahni [b]

[a] Adobe Systems Inc., Lehi UT, USA
[b] University of Florida, Gainesville FL, USA

ARTICLE INFO

Keywords:
Markov Decision Process
Value Iteration
Machine learning
Cache efficiency
Clustering
Annealing

ABSTRACT

Value Iteration (VI) is a powerful, though time consuming, approach to solve Markov Decision Processes (MDPs). Existing algorithms for VI incur a large number of cache misses. Motivated by the observation that, on modern computers, the cost of a cache miss is two to three orders of magnitude more than that of an arithmetic operation, we explore the possibility of improving the performance of VI by reducing the number of cache misses, possibly at the expense of increasing the number of arithmetic operations. Cache efficiency is obtained by performing VI on partitions of the MDP state space that fit in the lowest level cache of the computational platform on which the code is to run. Further performance improvement, motivated by the use of MDP partitions, is obtained using a clustering scheme to construct the partitions and an annealing schedule to converge to the target accuracy. We demonstrate experimentally that incorporating partitioning, clustering, and annealing into state-of-the-art VI software result in speedups of up to a factor of 8.1 on our computational platforms.

1. Introduction

Reinforcement Learning problems with Markov properties can be modeled as Markov Decision Processes (MDPs). There are a large number of real world applications of MDPs that have been well studied and implemented [1]. These applications come from diverse areas such as:

- Population (such as fish) harvesting
- Agriculture
- Purchasing, inventory and production
- Sales/Marketing promotions
- Patient admissions

Some examples of applications in the sales/marketing area [2] are:

- Decisions have to be made about price discounts and the duration of discounts that are to be offered on a product. The states represent the number of discounts currently running, discount periods remaining and current prices. The objective function is to maximize the expected profit over a finite horizon, where profit is computed as sales revenue discounting the penalty costs due to exceeding the budget.
- A common use case is deciding the size of an advertising budget. The states in this application are based on the budget spent in the past and resulting demand for the product. The reward is defined in terms of net profit, where profit is defined as sales revenue plus salvage values less advertising cost.

Value Iteration (VI) [3] is a powerful method for solving MDPs. However, algorithms based on VI can be extremely time consuming because of the large number of states in any practical problem and the large number of backups needed for VI to converge to the target accuracy. Many attempts have been made to improve the efficiency of VI [4–6]. These have focussed on reducing the number of backups by employing heuristics to eliminate redundant backups [4,5] and exploiting the graph structure of the MDP [4]. An approach to obtain an approximate solution of an MDP is described in [7]. Parallel solutions to solve MDP are also proposed in [8] and [9]. Efficient Algorithms for solving MDP under different kinds of budget constraints are proposed in [10]

To our knowledge, there has been no prior attempt to speed VI by exploiting the cache efficiency of modern computers. In this paper, we first develop a basic cache efficient VI algorithm that works on cache size partitions of the MDP state space. This basic cache efficient algorithm is then enhanced using clustering to construct the partitions and an annealing schedule to compute the values of all states with the target accuracy. By employing, partitioning, clustering, and annealing, we are able to attain a speed up of up to a factor of 7.88 relative to state-of-the-art VI software.

In Section 2 we provide a description of MDPs and VI. In Section 3, we describe a cache model that has been successfully used before

☆ A preliminary version of this work that does not include clustering and annealing appears in A. Jain and S. Sahni, Cache efficient value iteration, IEEE Symposium on Computers and Communications (ISCC), 2019.

* Corresponding author.
E-mail addresses: jaianuj99@gmail.com (A. Jain), sahni@cise.ufl.edu (S. Sahni).

https://doi.org/10.1016/j.comcom.2020.04.058
Received 26 November 2019; Received in revised form 23 March 2020; Accepted 30 April 2020
Available online 8 May 2020

in [11] to improve the performance of algorithms. This model provides the intuitive support for the base partitioning strategy that is developed in Section 4 to improve the cache efficiency of VI. In Section 5 we describe two strategies (clustering and annealing) to improve the performance of the VI algorithm that results from the application of the base partitioning strategy. An experimental evaluation of the performance impact of incorporating partitioning, clustering, and annealing into a state-of-the-art VI implementation that does not employ these strategies is provided in Section 6. We conclude in Section 7.

2. Markov decision processes and value iteration

A reinforcement learning task that satisfies the Markov Property is called a *Markov Decision Process* or *MDP*. An MDP comprises vertices and directed edges where each vertex represents a state and edges represent transitions from one state to another. When the agent is in a state *s*, it can choose from a set of possible actions. Having selected the action *a*, the agent moves into a next state *s'* with some probability. We use the term state space to represent set of states in the MDP. Action space refers the set of all actions. In a *finite MDP*, the state and action spaces are finite. Given any state, *s* and action, *a*, the probability of each possible next state, *s'* is given by Eq. (1).

$$p(s'|s,a) = Pr\{s_{t+1} = s'|s_t = s, a_t = a\}. \quad (1)$$

The goal of many reinforcement learning algorithms is to compute the value function of a problem, which is given by the *Bellman equation*:

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') \max_{a' \in A} Q(s',a') \quad (2)$$

where *Q(s,a)* denotes the value of taking action $a \in A$, *R(s,a)* is the immediate reward of taking action *a* from *s*, *T(s,a,s')* is the transition probability from *s* to *s'* using *a* and γ is the discount factor between 0 and 1. Qualitatively, the value function estimates *how good* it is for a reinforcement learning agent to be in a certain state or *(state,action)* pair. Therefore, optimal values would be the maximum possible values that converge for a given policy of the reinforcement learning agent. The same *Bellman equation* could also be expressed in terms of cost, where the problem is modeled such that the agent incurs an immediate cost for taking an action instead of getting a reward. This can be expressed as:

$$Q(s,a) = C(s,a) + \gamma \sum_{s'} T(s,a,s') \min_{a' \in A} Q(s',a') \quad (3)$$

In such a model the optimal values would be the minimal values that converge given a policy. Most optimal MDP algorithms are based on dynamic programing. The most powerful, yet simple dynamic programing algorithm is called **Value Iteration (VI)** (Bellman, 1957). In VI, the values of states or *(state, action)* pairs are updated iteratively in cycles, to obtain successively better approximations of the optimal values of each state, action (*s,a*) pair. In each cycle, the value of each state or *(state, action)* pair is updated using a **backup** operation in which we transfer the information *back* to a *state* or *(state, action)* pair from its successor *states* or *(state, action)* pairs. The VI algorithm stops when, we complete a cycle in which the value of each state changes by less than a specified threshold (ϵ_{final}) (Algorithm 1). This specified threshold is also called the *target accuracy*.

It is shown in [12] that the VI algorithm is guaranteed to converge regardless of the order in which updates are performed on the states. Prioritized VI (PVI) is a refinement of VI in which the state updates are done in priority order [3]. In PVI, each state is assigned a priority. To perform a backup, we select the state with highest priority to update. Following the update of this state, the priority values of dependent states are recomputed and the highest priority state becomes the next state to update. Since PVI incurs a high overhead to select the next state to update, Wingate et al. [6] developed a partitioned version to reduce this overhead. In their partitioned version, states are grouped into partitions and priorities are assigned to the whole partition. Subsequently VI is done on partitions in priority order. Wingate et al. [6] point out that although PVI works well with an appropriate choice for partition size, they do not know how to predict a good partition size. Our motivation to partition the state space is cache efficiency rather than reducing the overhead of selecting the next state to update. Good partition sizes in our scheme can be predicted based on cache size. Further, in our cache efficient algorithm, partitions are updated in a predetermined fixed order with no (dynamic) priority scheme employed.

Algorithm 1 Value Iteration

```
Input MDP
Initialize Q arbitrarily for each (state,action) pair.
while true do
    Bell_error ← 0
    for all s ∈ S do
        oldQ ← Q(s,a)
        Q(s,a) ← R(s,a) + γ Σ_{s'} T(s,a,s') max_{a'∈A} Q(s',a')
        Residual(s,a) ← |Q(s,a) − oldQ|
        Bell_error ← max(Bell_error, Residual(s,a))
    end for
    if Bell_error < ε_final then
        return Q
    end if
end while
```

3. Cache model

Since the time to access main memory is two to three orders of magnitude more than that required to perform an arithmetic operation, modern computers employ a limited amount of high speed memory (called cache) to hide memory latency. Although modern computers have several levels of cache (typically, L1, L2, and L3 with L3 being the largest and lowest-level cache), we employ a simple one-level cache model (the single level of cache may be assumed to be the lowest-level cache in the computer on which the code is to run) to provide intuitive support for the strategies proposed in this paper. As in prior work on cache efficiency [11], we assume the cache replacement policy is LRU (Least Recently Used). We assume that the cache consists of *c* cache lines. Each cache line consists of *w* words and each word consists of 4 bytes. So the total cache capacity is *cw* words. *w* is the number of words in a single cache line that can be transferred between the main memory and the cache in a single cache operation. Main memory is also organized in blocks of *w* words each. When a program needs to read a word that is not present in the cache then a cache miss occurs. To service the cache miss the block from the main memory containing the word is copied into the cache over the cache line selected using the LRU rule. Cache is written back with write allocate mechanism. Write allocate means that when we want to write a piece of data and it is not present in the cache then a write miss occurs. To service the write miss, the block of memory containing the word is copied into the cache from the main memory. Subsequently the data is written in the cache line only. It is written into main memory only when this cache line becomes the LRU cache line and needs to be overwritten by a block of fresh data from the main memory.

Therefore, every read and write miss makes a read access to the main memory. Also, every read and write miss makes a write access to the main memory when the LRU cache line about to be overwritten contains data that has been changed.

Modern computers use far more sophisticated cache replacement strategies than the simple LRU cache model described above. Some can even learn the memory access pattern and prefetch the data that need to be accessed into the cache. For our purposes here, a simple cache model, such as the one described above, is sufficient. Eventually the effectiveness of our cache efficient algorithm will be verified experimentally.

4. Base cache efficient VI

The value function used in VI has the form

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') \max_{a' \in A} Q(s',a') \quad (4)$$

Therefore, to update the value of a state–action pair for a state s, we need to access the current values of all possible successor states s' of s under all possible actions a. Since, for large MDPs, the cache size is much smaller than the memory required to hold the data associated with all MDP states, the update of the value of an (s,a) pair could incur several cache misses. This observation leads to the following strategies to improve cache efficiency possibly at the expense of an increase in the number of updates.

- **Partition the state space into sets of states, where each set is represented as *S***. As stated in [3] for the asynchronous dynamic programing algorithms used for evaluating an MDP, states in a given MDP can be updated in any order, using whatever values of the other states that happen to be available. Values of some states or (state, action) pairs may be backed up several times before some other states are backed up even once. The only condition required for the algorithm to converge is that no state is ignored unless it is known that backups of that state are not required to arrive at the optimal solution. Taking advantage of this flexibility in how backups may be done, we partition the state space into disjoint sets of states such that each set (and associated data) fits into cache. Then, instead of repeatedly performing backup cycles with every MDP state being backed up in a cycle, we begin by placing all partitions into a first-in-first-out queue. Next, we extract the first partition S from this queue and perform VI on the states in S (i.e., the state values in this partition are repeatedly backed up until $Q(s,a)$ for all $(s,a) \in S$ converge with an accuracy ϵ_{final}). If during the VI for S, the value of some state changes, then following the VI on S, all dependent partitions of S are added to the queue (unless they are already on the queue). We then work on the next partition in the queue. This continues until the queue becomes empty. Typically, when performing VI on a partition S, each state of S is visited multiple times. If all $s \in S$ together with all their properties fit into cache, then while performing VI on S, no write misses occur. The size of a partition is limited by the cache size and in our base cache efficient algorithm, states are assigned to partitions arbitrarily while ensuring that no partition exceeds the allowable maximum size.
- **Reduce read misses when performing VI on a partition S**. The partitioning process just described partitions the entire state space of the MDP into sets such that each state s belongs to exactly one S. Let SS be the set of all such sets. Let the complete state space be denoted by C.

$$\bigcup_{S \in SS} S = C \text{ and } S_i \cap S_j = \emptyset, i \neq j \quad (5)$$

Each $s \in S$ could transition into a state s' such that s' either belongs to the same partition S or to a different partition S'. The *external partitions* of S are as given by Eq. (6).

$$EP(S) = \{S' \mid \exists (s,a) \in S \wedge p(s' \in S' | s,a) > 0\} \quad (6)$$

The *dependent partitions* S'' of S are the partitions that contain at least one state s'' such that s'' can transition to a state $s \in S$ as in Eq. (7). Any change in value of $s,a \in S$ implies S'' needs to reevaluated.

$$DP(S) = \{S'' \mid \exists (s'',a'') \in S'' \wedge p(s \in S | s'',a'') > 0\} \quad (7)$$

Fig. 1 illustrates the concepts of external and dependent partitions. As noted earlier, write misses are eliminated during the VI of a partition S when S fits into cache. However, read misses may occur as the successor states for an $s \in S$ may be external to S. Whenever the value of such a successor state is needed we may incur a read miss. To avoid these read misses, we note that the value of external states does not change during the VI of S. So, the contribution of external states to the value of any state in S may be computed prior to beginning the VI of S and the aggregate contribution of all external states that an internal state s may transition to accounted for.

Algorithm 2 gives our base cache efficient VI algorithm that uses partitions and aggregation of external state values. This Algorithm is described in detail in [13]

Algorithm 2 Base Cache Efficient VI

1: Define the partition size based on the (lowest-level) cache size of the computational platform.
2: Partition the entire state space into sets of states such each set fits within the cache.
3: Add all these sets to a *queue*
4: If the *queue* is empty, terminate the algorithm, else remove the first Set S from the *queue* and choose it as the current set for performing VI
5: Load the values of all the states external to S that would be required for performing VI as some states from S may transition to these external states $s'_e \notin S$.
6: Aggregate the impact of external states $s'_e \notin S$ for every (state,action) pair $(s,a) \in S$ as in: $e.c_{s,a} = \gamma \sum_{s'_e} T(s,a,s'_e) S.ext\_sets[S'].ext\_states[s'_e]$
7: Perform VI on all $(s,a) \in S$ until convergence, using the following: $Q(s,a) \leftarrow R(s,a) + \gamma \sum_{s'_i} T(s,a,s'_i) \max_{a' \in A} Q(s'_i,a') + e.c_{s,a}$
8: Finally, after S converges, add any dependent sets S'' to the *queue* and go back to step 4

For Algorithm 2, we note that:

- The complete information $\forall\ s \in S$ should be loaded into cache prior to performing VI on S. To do this efficiently, we store the information required for each S in contiguous memory.
- Ideally we want that once the information about all the states $s \in S$ is loaded into cache and we start performing VI on these states, we do not have to service a cache miss until convergence. Therefore, the values of all the external states (s') should also be present in cache. Hence, as described in steps 5 and 6 of Algorithm 2 we load the values of all the external states required to perform this VI even before we begin the VI on the set S.
- Notice that in step 7, while performing VI on S, which may include several thousands of operations, we can use a precomputed impact of all the external states on a given (s,a) pair. This is the value $e.c_{s,a}$. This is because the value of these external states does not change while performing VI on S and hence their aggregate impact does not change either. Therefore, it is feasible to precompute the aggregate impact of external states and cache it till the convergence of S.

5. Enhancements to the base algorithm

In Section 4 we described our base cache efficient VI algorithm. In this section we introduce two strategies to enhance the performance of this base algorithm.

- **Clustering**. When states within a given partition have dependencies across partition boundaries, extra work needs to be done to cache the values of such states. Further, whenever the value of a state $s \in S$ changes and s has dependencies in a set $S' \neq S$, then S' may need to be scheduled again for performing VI on it as described in Algorithm 2. To reduce these side effects, we

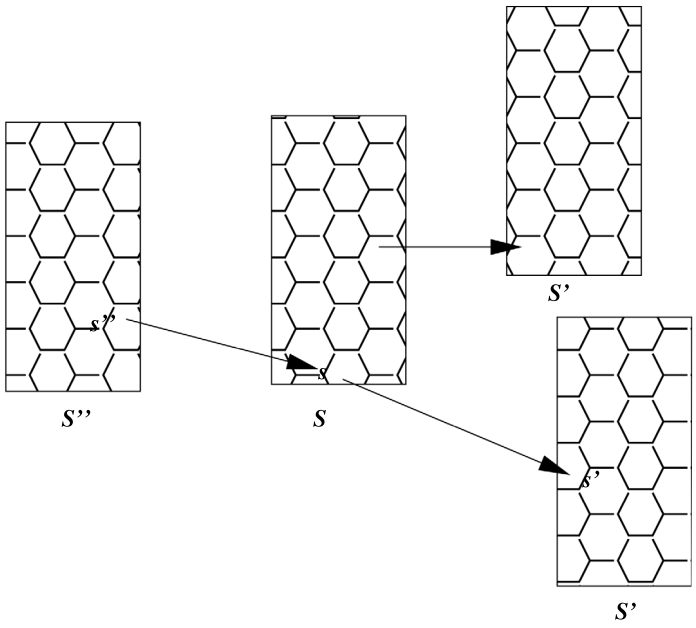

Fig. 1. Example of arbitrary sets with states s'', s and s'. $s \in S$ can transition to $s' \in S'$, therefore S' is an external partition for S. S'' is dependent on S as $s'' \in S''$ can transition to $s \in S$.

use a clustering scheme to construct partitions with fewer inter-dependencies of states across partition boundaries. To minimize the overhead of the clustering scheme, we use a simple heuristic to partition the MDP such that highly connected states fall into the same partition. Starting from an arbitrary state s, for each possible action a from s, we perform a breadth first search (BFS), such that all possible next states s' that have high transition probability are included in the same partition S as the state s, until we reach the pre-defined maximum size of the partition (this maximum is based on the cache size of the hardware platform). The transition probability to a next state s' from s given action a is considered high if it adds at least a pre-defined threshold percentage to the total probability accumulated for transitioning out of s given a.
In our base plus clustering VI algorithm, Step 2 of the base algorithm (Algorithm 2) is replaced by the steps:

- 2a: Repeat Step 2b until all MDP states have been assigned.
- 2b: Pick an arbitrary unassigned state s and assign it to a new set/partition S. Perform a BFS beginning at s. Unassigned states reached during this BFS are assigned to S if they make a reasonable contribution (see below) to the total transition probability of the state currently being examined by the BFS. Stop the BFS when S reaches the permissible partition size or no unassigned reachable state remains.

Specifically, when the BFS of Step 2b above examines an as yet unexamined state t that is in S, the following is done.

- Repeat the following steps for each action a possible at state t.
- Sort the unassigned states that can be transitioned to from t by performing the action a into descending order of transition probabilities. Let $t_1, \ldots, t_k$ be the sorted list of states that can be transitioned to.
- Add t_1 to S and set $TotPr(t, a) = Pr(t_1|t, a)$. Set $i = 2$.
- While ($i <= k$ and $Pr(t_i|t, a) > 0.2 * TotPr(t, a)$) { Add t_i to S; $TotPr(t, a) += Pr(t_i|t, a)$; $i++$}

- **Annealing**. In the annealing of a hot metal, the hot metal is cooled gradually to get it to a desired state. The concept of gradual cooling may be employed to our base cache efficient VI algorithm with a view to reducing run time. In the base algorithm, whenever a partition is worked on, VI is used until no state sees a change in value larger than the target accuracy ϵ_{final}. The rationale behind this is that while a partition is in the cache we want to maximize the work done on it toward achieving the target accuracy. However, the convergence to the target accuracy is done using values of as yet unconverged external dependant states. When the values of these dependent states change as a result of performing VI on their partitions, we have to redo VI on the partition that is currently in the cache. In an annealing approach to achieve global convergence to the target accuracy, we set up an annealing schedule $\epsilon_1 > \epsilon_2 > \cdots > \epsilon_k = \epsilon_{final}$ and run the base algorithm k times seeking first global convergence to ϵ_1, then to ϵ_2, ⋯, and finally to the target ϵ_{final}. So, like in the annealing of a hot metal, we converge to the target state defined by ϵ_{final} gradually. Through experimentation, we discovered that better performance is achieved by modifying the described annealing strategy that anneals all partitions at the same rate to one that anneals different partitions at possibly different rates (i.e., partition localized annealing). When partitions are initially placed into the first-in-first-out queue of Algorithm 2, each is assigned a convergence value of ϵ_1. When a partition is removed from this queue an initially zero counter local to the partition is incremented by 1, VI is performed on that partition using its current convergence value, and if the partition's local counter is a multiple of 10, the partition's convergence value is changed to the next epsilon (if any) in the annealing schedule.
We note that our adaptation of annealing has similarities to the combinatorial optimization method of simulated annealing [14] in its use of an annealing schedule. Our adaptation differs from simulated annealing in that we do not use the annealing schedule to decide whether or not to accept bad moves (i.e., moves that take you away from the desired final state); in simulated annealing, the annealing schedule is used to determine the probability with which a bad move is to be accepted. In fact, we do not permit any bad moves. Further, our use of an annealing schedule does not affect the final state values computed by our adaptation of VI, whereas in simulated annealing, the final results vary with the annealing schedule in use.

Algorithm 3 is our final cache efficient VI algorithm that incorporates clustering and partition localized annealing. This algorithm

uses the annealing schedule $\{10, 1, 0.1, \ldots, 10^{-6}\}$ (this schedule may be changed to any other suitable schedule). So, $\epsilon_1 = 10$ and $\epsilon_{final} = 10^{-6}$. We begin by using clustering to partition the states of the MDP into disjoint sets each of which fits into the cache of the computer on which the code is to run. Since modern computers have multiple levels of cache, we set the allowable partition size based on the size of the lowest level (typically L3) cache in the target computer. Next, in lines 2 to 9, we do the setup, where for each set S we identify the external partitions and states *(S',s')* as well as the dependent partitions *S''*. We store this information in appropriate data structures associated with each set S and initialize a *queue* with all the partitions. Subsequently, in lines 11 to 32, we pick up partitions from the *queue*, one at a time, and perform VI on the picked partition. While performing VI on a partition, we read the values of the external states only once before the first iteration and compute the contribution of all such external states to *(s,a)* in $ExtSetSt(s, a)$. This way we do not incur a cache miss while performing any of the subsequent iterations of VI on the partition, until convergence. Also, as described in step 14, to read the values of the external states we go in the order of the external sets that these states belong to. Therefore, we only incur as many cache misses as the number of external sets *S'* for any given set *S*. Finally if there is any change in a value of a state $s \in S$ we add the dependents sets *S''* to the *queue*. The *q.enqueue*() operation is designed such that it adds an item to the queue only if not already present. The accuracy (ϵ) to use when performing VI on a partition S is given by $S.\epsilon$. This is changed to the next epsilon (if any) by dividing by 10 whenever its counter $S.counter$ becomes a multiple of 10 (line 28).

Algorithm 3 Cache Efficient Value Iteration With Clustering and Annealing

1: Use clustering to partition the state space into sets of a size that fit into the cache. Let SS be the set of partitions obtained.
2: **for all** $S \in SS$ and $(s, a) \in S$ **do**
3: **for all** $(s' \mid p(s'|s,a) > 0$ and $s' \notin S)$ **do**
4: Add (S', s') to $ExtSetSt(S, s, a)$
5: Add S to $S.D(S')$
6: **end for**
7: $S.\epsilon = 10$; $S.counter = 0$;
8: *q.enqueue(S)*
9: **end for**
10: $\epsilon_{final} = 10^{-6}$
11: **while** !*q.isEmpty()* **do**
12: $S \leftarrow$ *q.deque()*
13: **for all** *(S',s')* $\in ExtSetSt(S, s, a)$ **do**
14: $ExtSetSt(Ssa) \leftarrow V[s']$ ▷ *V[s']* $= \max_{a' \in A} Q(s', a')$
15: **end for**
16: **for all** *(s,a)* $\in S$ **do** ▷ Sum external state contributions
17: $e.c_{s,a} \leftarrow \gamma \sum_{s'} T(s, a, s') ExtSetSt(S, s, a)$
18: **end for**
19: Perform VI on S with $\epsilon = S.\epsilon$ using eqn:
20: *Q(s,a)* $\leftarrow$ *R(s,a)* +
21: $\gamma \sum_{s'_i} T(s, a, s'_i) \max_{a' \in A} Q(s'_i, a') + e.c_{s,a}$ ▷ $s'_i \in S$
22: $S.counter++$
23: **if** $Q(s, a)$ for any (s,a) has changed **then**
24: *q.enqueue(S.D(S))* ▷ Add dependent sets to queue
25: **if** $S.\epsilon > \epsilon_{final}$ **then**
26: *q.enqueue(S)* ▷ Add the set back to queue
27: **if** $S.counter\%10 == 0$ **then**
28: $S.\epsilon = S.\epsilon/10$ ▷ Next epsilon
29: **end if**
30: **end if**
31: **end if**
32: **end while**

6. Experimental results

To evaluate the effectiveness of partitioning, clustering, and (partition localized) annealing we incorporated these into the state-of-the-art VI algorithm $FTVI$ [4]. We chose $FTVI$ for our experiments because experimental results reported in [4] indicate that $FTVI$ outperforms other known VI algorithms. A high level description of the $FTVI$ algorithm, as described in [4], is provided in Algorithm 4. Let BCE (Base Cache Efficient), CEC (Cache Efficient with Clustering), and $CECA$ (Cache Efficient with Clustering and Annealing), respectively, be $FTVI$ modified to invoke our base cache efficient algorithm (Algorithm 2), our cache efficient with clustering algorithm, and our cache efficient with clustering and annealing algorithm (Algorithm 3), for SCCs (strongly connected components) whose size (i.e., number of states) exceeds a threshold, which was set to 1000 states in our experiments.

Algorithm 4 High Level Description of FTVI

1: Perform Search operation on the state space to eliminate sub-optimal transitions. This reduces the connectivity of the graph.
2: Partition the state space graph into strongly connected components ($SCCs$). This also orders the components in reverse topological order from goal state/s to start state/s.
3: Perform VI on the $SCCs$ in reverse topological order.

To compare the performance of $FTVI$, BCE, CEC, and $CECA$ we used the following data sets [4,5]:

(1) Mountain Car (MCAR) — This is a two-dimensional control problem that is characterized by position and velocity. A small car must rock back and forth until it gains enough momentum to carry itself up the top of the hill. Any exit on the left hand side of the problem results in a reward of −1. A gradient reward is given on the right hand side, with the maximum reward of 1 being given if the car exits the state space with zero velocity. A high velocity results in a reward of −1.
(2) Single Arm Pendulum (SAP) — This is a two dimensional minimum time optimal control problem. The agent has two actions available representing positive and negative torques applied to a rotating pendulum, which the agent must learn to swing up and balance. Similar to the MCAR the agent cannot move the pendulum from the bottom to the top directly, but must learn to rock it back and forth. Rewards are zero everywhere but in the balanced region. The state space is defined by the angle of the link (θ) and the angular velocity of the link ($d\theta/dt$).
(3) Double Arm Pendulum (DAP) — This is a four-dimensional minimum-time control problem. A central motor applies torque to the primary link. It is similar to SAP, except that there are two links. The agent must balance the second link vertically but it is a free-swinging link. The state space is defined by the two linkages (θ_1, θ_2) and their angular velocities ($d\theta_1/dt$, $d\theta_2/dt$).

The MDPs for MCAR, SAP, and DAP were generated using the data generation programs of [5]. For MCAR, we generated two instances. One of these has 1M states and the other has 4M states. For DAP we also generated two instances. One of these has 1M states and the other has 2M states. For SAP, we used only a single instance with 0.8M states. We were unable to use larger SAP instances as the FTVI code crashed on instances with more states. While we experimented also with the Mesh problem of [5], the Mesh MDPs had no SCC whose size exceeded the threshold of 1000 states used to invoke BCE, CEC, and $CECA$. As a result, $FTVI$, BCE, CEC, and $CECA$ had the same performance on Mesh instances. We were unable to use the remaining data sets of [4] as the generators for these data sets were not available. Since $FTVI$ requires that the MDP have one or more goal states specified, our data

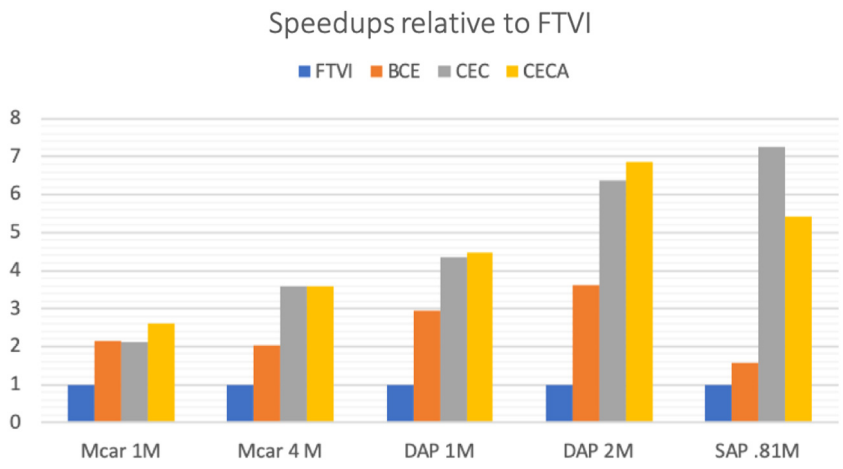

Fig. 2. Platform 1.

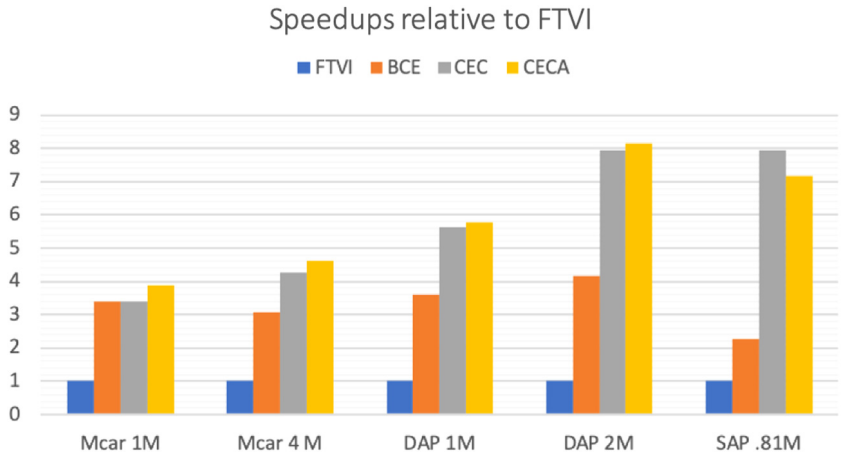

Fig. 3. Platform 2.

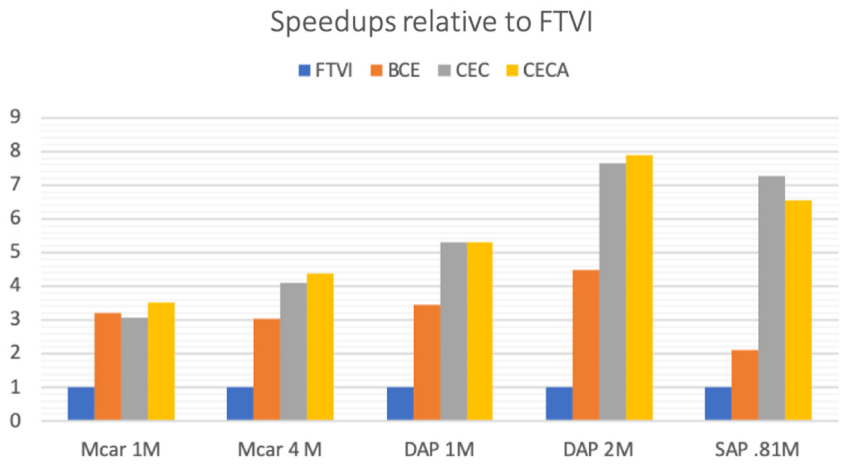

Fig. 4. Platform 3.

sets also had goal states. So, the experiments reported in the sequel involve a total of 5 instances.

Our experiments were performed on three different computational platforms:

- Platform 1: MAC with an Intel Core, base clock frequency 2.5 GHz i7 Processor with 3 levels of cache and 16 GB of main memory. The cache sizes: *L1 Cache:* 32 KB, *L2 Cache:* 256 KB, *L3 Cache:* 6 MB.
- Platform 2: PC with Intel Core, base clock frequency 3.3 GHz, i9 Processor with 3 levels of Cache and 64 GB main memory. The cache sizes: *L1 Cache:* 32 KB, *L2 Cache:* 1 MB, *L3 Cache:* 14 MB
- Platform 3: PC with Intel (R) Xeon (R) Platinum 8151 Core, base clock frequency 3.4 GHz, Instance type Z1d.metal from AWS with

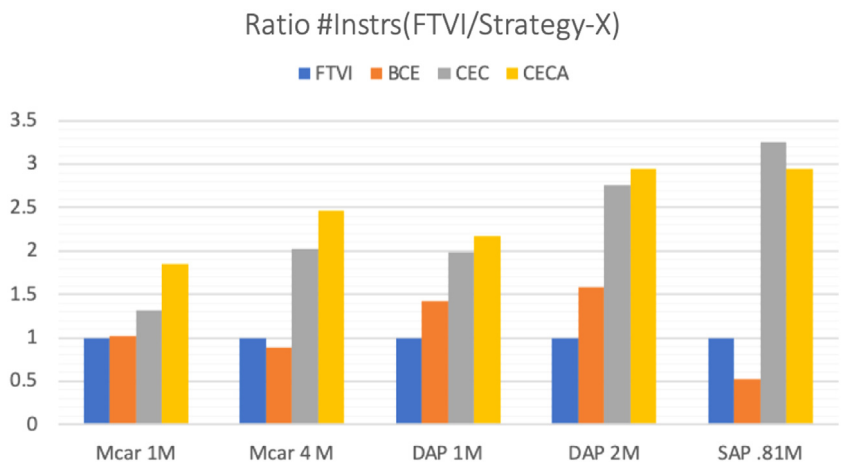

Fig. 5. Platform 1.

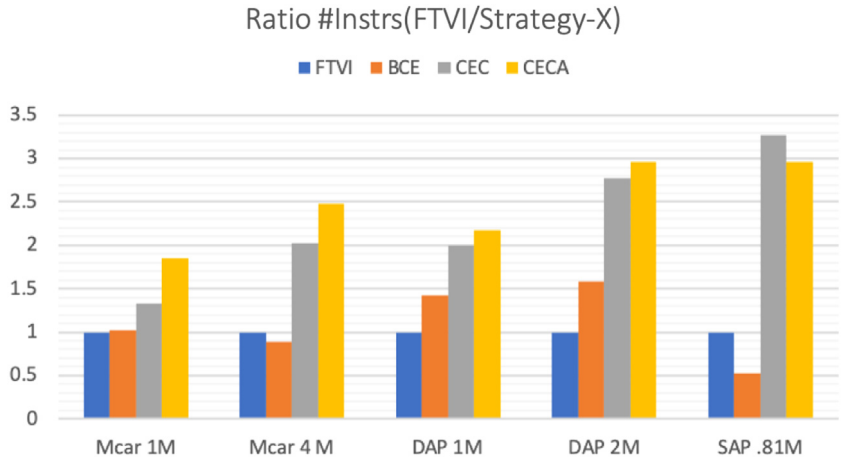

Fig. 6. Platform 2.

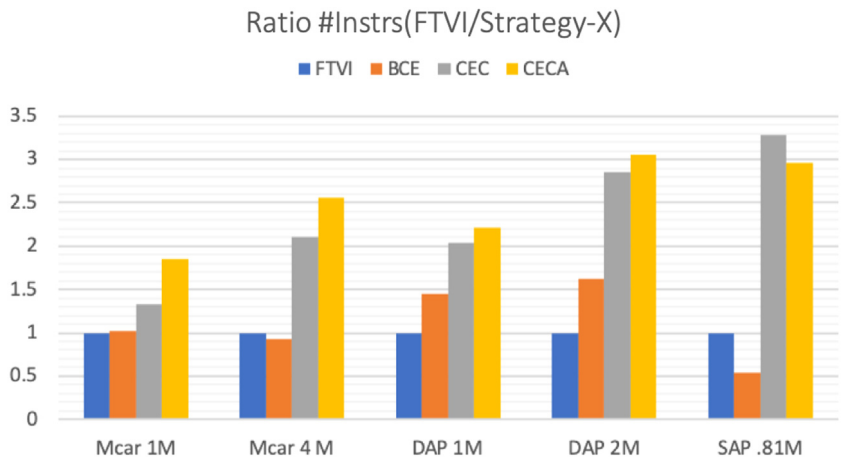

Fig. 7. Platform 3.

3 levels of Cache and 300 GB. The cache sizes: *L1 Cache:* 32 KB, *L2 Cache:* 1 MB, *L3 Cache:* 25 MB

We first conducted exploratory experiments on Platform 1 to determine a good partition size to use. These experiments were guided by the size (6MB) of the lowest level cache (L3). For Platform 1, partitions of size 5000 states fit into L3 cache and gave best performance for BCE. In addition to the reduction in cache misses that were expected from the use of our partitioning strategy, the exploratory experiments revealed a significant drop in the number of backups and, consequently, in the number of instructions executed. In the extreme case when the whole MDP fits into L3 cache, there is only 1 partition and no reduction in backups. So, while larger partitions (up to the size of L3 cache) improve cache behavior, larger partitions negatively impact the number of backups and, in turn, result in longer run times. as a result, BCE, gave best performance on our remaining platforms also

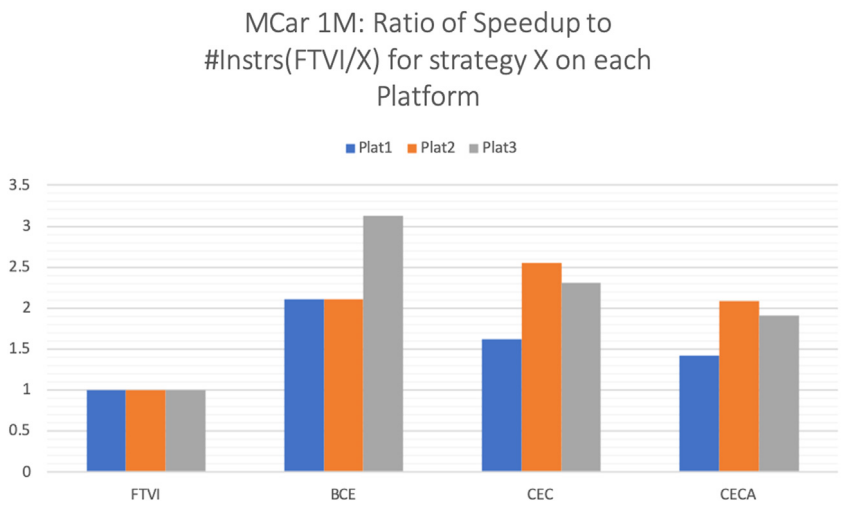

Fig. 8. MCar 1 M.

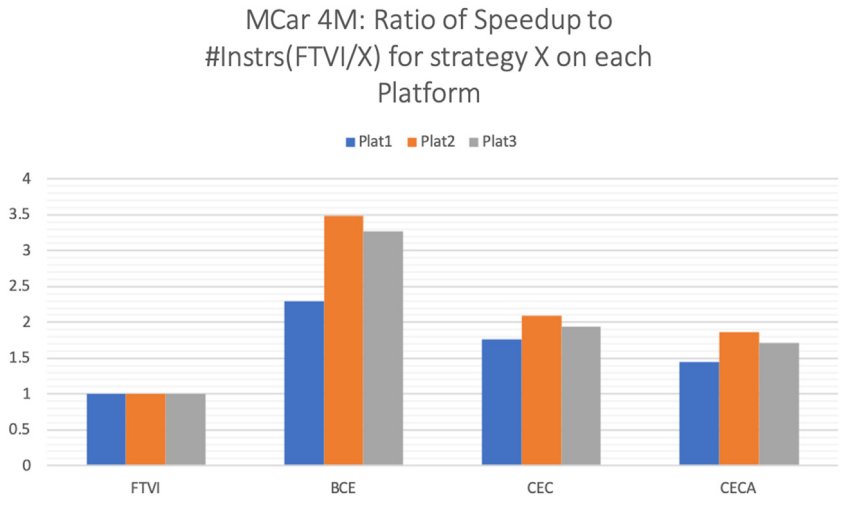

Fig. 9. Mcar 4M.

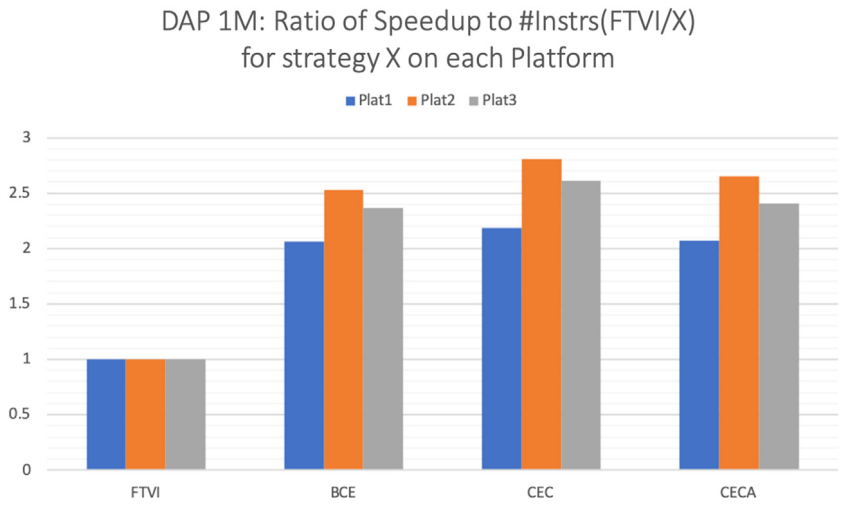

Fig. 10. DAP 1M.

using a partition size of 5000 even though these platforms have a larger L3 cache. For CEC and $CECA$, best run time performance was observed using the smaller partition size of 1300. This is because, for these methods, the advantage of a smaller number of backups resulting from the smaller partition size outweighed the benefits from improved cache utilization using a larger partition size. The experimental results presented in this section use a partition size of 5000 for BCE and 1300 for CEC and $CECA$.

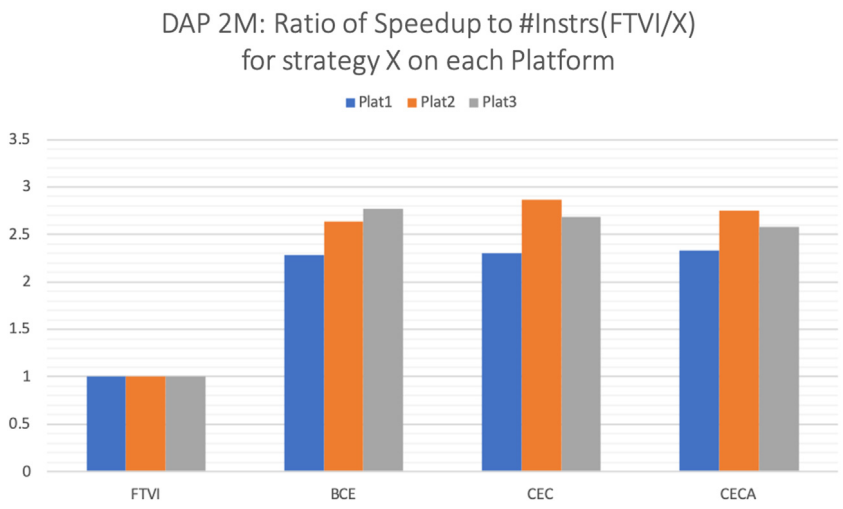

Fig. 11. DAP 2M.

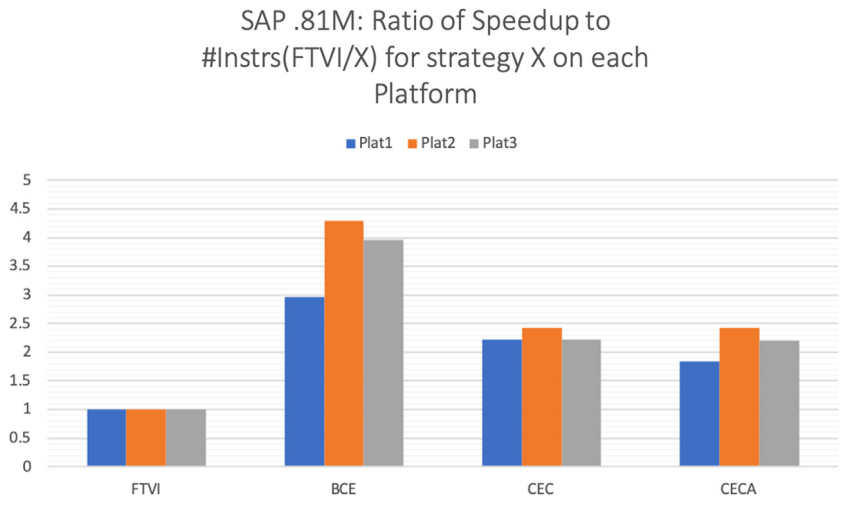

Fig. 12. SAP .81M.

For each data set, we measured the total time to solve the MDP (this includes the time for searching through the MDP and for partitioning the MDP into components, which is common across all the algorithms being evaluated), the time to solve the largest SCC (each data set had only one SCC whose size exceeded the threshold size of 1000 used by us), the number of backups (i.e., the number of state updates) to solve all SCCs, total cache misses, total number of micro instructions executed by the programs, and the cache misses per instruction. Also, reported is the number of instructions executed per cycle for each of the programs. All three computational platforms ran the Ubuntu operating system and all codes were written in C++. The cache misses and all the instruction level data was obtained using the "perf" [15] tool. The maximum difference in the computed values for the *(s,a)* pairs by the four different algorithms across the entire state space after convergence was less than 10^{-6} for all our data sets. The threshold, ϵ_{final}, for VI convergence was set to 10^{-6} for all runs. For annealing, the schedule $\{10, 1, 0.1, \ldots, 10^{-6}\}$ was used.

Tables 1–15 give the experimental results for our 5 data sets and 3 computational platforms. Figs. 2–4 give the speedups attained by BCE, CEC, and $CECA$ relative to $FTVI$ (i.e., $runtime(FTVI)/runtime(X)$, $X \in \{BCE, CCE, CCEA\}$) on each of our 3 computational platforms. This speedup ranges from a low of 1.56 for the 0.81M state SAP problem run on platform 1 with BCE Algorithm to a high of 8.15 for the $2M$ state DAP problem run on platform 2 with $CECA$ Algorithm. So, on all instances and platforms BCE, CEC, and $CECA$ were faster than $FTVI$. CEC was faster than BCE in all cases except on Mcar 1M problem where they are almost similar. Barring these exceptions, the speedup attained by CEC relative to BCE ranged from a low of 1.72 for the 4M state MCar problem run on platform 1 to a high of 4.61 for the $.81M$ state SAP problem run on platform 1. $CECA$ was faster than CEC in all cases except SAP with 0.81M states on all platforms. Barring these exceptions, the speedup attained by $CECA$ relative to CEC ranged from a low of 1.005 for the 1M state DAP problem run on platform 3 to a high of 1.5 for the 1M state Mcar problem run on platform 1.

In all cases, BCE, CEC, and $CECA$ had fewer cache misses and backups than incurred by $FTVI$. The instruction count was also usually less. The instruction count increased for BCE on MCAR with 1M states and 4M states. It also increased for BCE for SAP with 0.81M states on all Platforms. Figs. 5–7 plot the ratio $\#Instrs(FTVI)/\#Instrs(X)$ for $X \in \{BCE, CEC, CECA\}$. This ratio ranges from a low of 0.52 to a high of 3.28. Despite the increase in instruction count on some instances, run time decreased indicating the speedup came from a reduction in cache misses. We can get insight into the contribution of reduced cache misses to the reduction in run time by examining the values in the rows labeled #Instructions/CPU-cycle in the tables as well as the ratio (speedup relative to FTVI) divided by $(\#instructions(FTVI)/\#instructions(X))$ for $X \in \{BCE, CEC, CECA\}$. The latter ratio, which is shown in Figs. 8–12 for each of our 3 computational platforms, ranges from a low of 1.3 to a high of 4.3

Table 1
Platform 1: Mountain Car problem with 1 M states. Size of the largest component: 767098 states. (s = seconds, M = million, B = billion).

Platform 1				
Metric	MCar 1 M States			
	BCE	*CEC*	*CECA*	*FTVI*
Run time	30.72 s	30.92 s	25.14 s	65.86 s
Time for Biggest Comp	22.6 s	22.8 s	16.6 s	61.81 s
Backups	276.7 M	213M	121M	600 M
#Instructions	210 B	162.4B	115.9B	214 B
Cache-misses	226.2M	337.4M	355M	945M
Cache-misses/ Instruction	0.10%	0.2%	0.3%	0.44%
#Instructions/ CPU-cycle	1.85	1.6	1.25	0.88

Table 2
Platform 1: Mountain Car problem with 4 M states. Size of the largest component: $3.18M$ states.

Platform 1				
Metric	MCar 4 M States			
	BCE	*CEC*	*CECA*	*FTVI*
Run time	284.2 s	161.19 s	148 s	577.5 s
Time for Biggest Comp	218.36 s	114.94 s	101 s	515.6 s
Backups	2.8 B	1.09B	770M	5.2 B
#Instructions	1.95 T	854B	700.2B	1.73 T
Cache-misses	1.8 B	2.8B	2.86B	8.5 B
Cache-misses/ Instruction	0.05%	0.3%	0.4%	0.5%
#Instructions/ CPU-cycle	1.88	1.36	1.17	0.82

clearly indicating that much of the speedup comes from increased cache efficiency. We note that under the assumption that the instruction mix has not changed, this ratio would be very close to 1 if almost all the speedup came from a reduction in the number of instructions executed.

Comparing BCE and CEC, Clustering is a heuristic and gives less number of cache misses on all problem sets on Platform 2 and 3. There are exceptions on platform 1, however. In all cases it reduces the number of backups by sufficient amount that there was a noticeable reduction in run time in all test cases. Comparing $CECA$ with the other two techniques, annealing is a heuristic that was used to reduce the number of backups and consequently number of instructions at slight cost of the cache efficiency. In all problem sets except the SAP problem instance, annealing reduces the number of backups and number of instructions.

The cache miss rate for BCE is similar for platforms 2 and 3 and is much lower for platform 1. Since different architectures use different cache replacement strategies, we expect variations in the number of cache misses per instruction. Using our simple cache model, our algorithms reduce run time on all platforms on all problem sets.

7. Conclusion

We have proposed several techniques (Partitioning, Clustering, Annealing) to speed the performance of classical implementations of VI and benchmarked these against the state-of-the-art $FTVI$ code. The effectiveness of our techniques was demonstrated experimentally and speedups of up to 8.1 relative to a classical implementation attained. Although, some of the speedup comes from a reduction in the number of instructions executed, not all of it does as the ratio (speedup relative to FTVI)/(#$instructions(FTVI)$/#$instructions(X)$) for $X \in \{BCE, CEC, CECA\}$ ranged from a low of 1.3 to a high of 4.3. If almost all the speedup came from a reduction in the number of instructions executed, this ratio would be close to 1 (assuming the instructions mix has not changed). Although BCE was motivated

Table 3
Platform 1: Double Arm Pendulum problem with 1 M states. Size of the largest component: $0.9M$ states.

Platform 1				
Metric	DAP 1 M States			
	BCE	*CEC*	*CECA*	*FTVI*
Run time	74.5 s	48 s	46 s	219.64 s
Time for Biggest Comp	56.24 s	36.12 s	34.12 s	201.90 s
Backups	286M	220M	178M	1.21B
#Instructions	360B	258B	237B	513.7B
Cache-misses	1.3B	980M	997.5M	5.1B
Cache-misses/ Instruction	0.36%	0.37%	0.4%	1%
#Instructions/ CPU-cycle	1.32	1.39	1.25	0.64

Table 4
Platform 1: Double Arm Pendulum problem with 2 M states. Size of the largest component: $1.9M$ states.

Platform 1				
Metric	DAP 2 M States			
	BCE	*CEC*	*CECA*	*FTVI*
Run time	159.94 s	90.8 s	84.1 s	577.36 s
Time for Biggest Comp	126.4 s	49.8 s	44.3 s	556.94 s
Backups	823 M	415MM	351M	3.4B
#Instructions	886.6B	508.6B	475.6B	1.4T
Cache-misses	2.6 B	1.86 B	1.87	12.2B
Cache-misses/ Instruction	0.29%	0.36%	0.39%	0.85%
#Instructions/ CPU-cycle	1.51	1.52	1.53	0.67

Table 5
Platform 1: Single Arm Pendulum problem with 0.81 M states. Size of the largest component: $0.8M$ states.

Platform 1				
Metric	SAP 0.81 M States			
	BCE	*CEC*	*CECA*	*FTVI*
Run time	74.72 s	16.12 s	18.48 s	116.92 s
Time for Biggest Comp	69.5 s	14.16 s	16.4 s	111.57 s
Backups	1.02B	108M	101M	1.07B
#Instructions	661.5B	107.1B	118.5B	348.8B
Cache-misses	113 M	240M	278M	1.63B
Cache-misses/ Instruction	0.017%	0.2%	0.23%	0.46%
#Instructions/ CPU-cycle	2.41	1.81	1.49	0.81

Table 6
Platform 2: Mountain Car problem with 1 M states. Size of the largest component: 767098 states. (s = seconds, M = million, B = billion).

Platform 2				
Metric	MCar 1 M States			
	BCE	*CEC*	*CECA*	*FTVI*
Run time	29.2 s	29.3 s	25.6	98.98 s
Time for Biggest Comp	20.01 s	20.19 s	16.36 s	89.72 s
Backups	276.7 M	214.3M	121.7M	600 M
#Instructions	210 B	161.3B	115.3B	213.8 B
Cache-misses	1.05B	564M	588M	10.3 B
Cache-misses/ Instruction	0.5%	0.34%	0.5%	5%
#Instructions/ CPU-cycle	1.69	1.29	1.06	0.51

Table 7
Platform 2: Mountain Car problem with 4 M states. Size of the largest component: $3.18M$ states.

Platform 2				
Metric	MCar 4 M States			
	BCE	*CEC*	*CECA*	*FTVI*
Run time	278.3 s	201.3 s	186.2 s	856.6 s
Time for Biggest Comp	189.14 s	118.2 s	104.12 s	773.3 s
Backups	2.8 B	1.09B	770M	5.2 B
#Instructions	1.95 T	848B	695.8B	1.72 T
Cache-misses	12.8 B	4.6B	4.7B	91.1 B
Cache-misses/ Instruction	0.63%	0.5%	0.6%	5.3%
#Instructions/ CPU-cycle	1.65	1.0	0.88	0.47

Table 8
Platform 2: Double Arm Pendulum problem with 1 M states. Size of the largest component: $0.9M$ states.

Platform 2				
Metric	DAP 1 M States			
	BCE	*CEC*	*CECA*	*FTVI*
Run time	88.9 s	57.11 s	55.5 s	320.9 s
Time for Biggest Comp	67.8 s	38.1 s	36.5 s	301.90 s
Backups	286M	220M	178.9M	1.21B
#Instructions	360B	256.9B	235.9B	513.7B
Cache-misses	2.7B	1.6B	1.6B	33.46B
Cache-misses/ Instruction	0.75%	0.6%	0.67%	6.5%
#Instructions/ CPU-cycle	0.95	1.05	1.0	0.38

Table 9
Platform 2: Double Arm Pendulum problem with 2 M states. Size of the largest component: $1.9M$ states.

Platform 2				
Metric	DAP 2 M States			
	BCE	*CEC*	*CECA*	*FTVI*
Run time	209.9 s	110.2 s	107.3 s	875 s
Time for Biggest Comp	165.3 s	72.7 s	69.8 s	837.9 s
Backups	823 M	405M	351M	3.4B
#Instructions	886.3B	505.7B	472.6B	1.4T
Cache-misses	6 B	3.32 B	3.34B	97.7B
Cache-misses/ Instruction	0.67%	0.65%	0.7%	6.9%
#Instructions/ CPU-cycle	0.99	1.08	1.04	0.38

Table 10
Platform 2: Single Arm Pendulum problem with 0.81 M states. Size of the largest component: $0.8M$ states.

Platform 2				
Metric	SAP 0.81 M States			
	BCE	*CEC*	*CECA*	*FTVI*
Run time	66.32 s	18.9 s	20.9 s	149.88 s
Time for Biggest Comp	61.4 s	13.54 s	15.55 s	144.58 s
Backups	1.02B	106M	119M	1.07B
#Instructions	661.3B	106.4B	117.7B	348B
Cache-misses	1.03 B	650M	741M	17B
Cache-misses/ Instruction	0.15%	0.6%	0.62%	4.8%
#Instructions/ CPU-cycle	2.3	1.34	1.34	0.55

Table 11
Platform 3: Mountain Car problem with 1 M states. Size of the largest component: 767098 states. (s = seconds, M = million, B = billion).

Platform 3				
Metric	MCar 1 M States			
	BCE	*CEC*	*CECA*	*FTVI*
Run time	30.7 s	32.23 s	27.98 s	98.56 s
Time for Biggest Comp	21.1 s	21.2 s	17.05 s	89 s
Backups	276.7 M	213M	121.4M	600 M
#Instructions	211.6 B	163.5B	117.3B	217.1 B
Cache-misses	1.07B	575M	598M	10.4 B
Cache-misses/ Instruction	0.5%	0.35%	0.5%	4.7%
#Instructions/ CPU-cycle	1.72	1.27	1.05	0.55

Table 12
Platform 3: Mountain Car problem with 4 M states. Size of the largest component: $3.18M$ states.

Platform 3				
Metric	MCar 4 M States			
	BCE	*CEC*	*CECA*	*FTVI*
Run time	281.98 s	208.4 s	195.08 s	852.4 s
Time for Biggest Comp	194.14 s	136.8 s	122.3 s	768.3 s
Backups	2.8 B	1.09B	770M	5.2 B
#Instructions	1.96 T	861.3B	707.8B	1.81 T
Cache-misses	12.9 B	4.8B	4.85B	93.5 B
Cache-misses/ Instruction	0.65%	0.55%	0.68%	5.1%
#Instructions/ CPU-cycle	1.75	1.03	0.91	0.53

Table 13
Platform 3: Double Arm Pendulum problem with 1 M states. Size of the largest component: $0.9M$ states.

Platform 3				
Metric	DAP 1 M States			
	BCE	*CEC*	*CECA*	*FTVI*
Run time	89.17 s	57.63 s	57.6 s	306.7 s
Time for Biggest Comp	70.38 s	38.7 s	39.1 s	287.8
Backups	286M	220M	179M	1.21B
#Instructions	363.3B	259B	238.3B	527.9B
Cache-misses	2.8B	1.58B	1.6B	31.9B
Cache-misses/ Instruction	0.77%	0.6/%	0.67%	6.0%
#Instructions/ CPU-cycle	1.02	1.12	1.03	0.43

Table 14
Platform 3: Double Arm Pendulum problem with 2 M states. Size of the largest component: $1.9M$ states.

Platform 3				
Metric	DAP 2 M States			
	BCE	*CEC*	*CECA*	*FTVI*
Run time	195.77 s	114.9 s	111.83	880.8 s
Time for Biggest Comp	157.9 s	74.7 s	71.1 s	842.7 s
Backups	823 M	415M	351M	3.4B
#Instructions	900B	511B	478.5B	1.46T
Cache-misses	6.2 B	3.3 B	3.34B	96.2B
Cache-misses/ Instruction	0.68%	0.64%	0.69%	6.5%
#Instructions/ CPU-cycle	1.15	1.11	1.07	0.42

Table 15
Platform 3: Single Arm Pendulum problem with 0.81 M states. Size of the largest component: $0.8M$ states.

Platform 3				
Metric	SAP 0.81 M States			
	BCE	*CEC*	*CECA*	*FTVI*
Run time	71.26 s	20.7 s	23 s	150.77 s
Time for Biggest Comp	64.9 s	14.4 s	16.7 s	145.25 s
Backups	1.52B	106M	119M	1.07B
#Instructions	662.4B	107.6B	119.1B	353.4B
Cache-misses	1.05 B	638.6M	730M	17B
Cache-misses/ Instruction	0.15%	0.59%	0.61%	4.8%
#Instructions/ CPU-cycle	2.33	1.30	1.3	0.59

by a simple one-level cache model using the LRU cache replacement strategy, as in [11], it results in significant performance improvement on tested contemporary computers that have 3 levels of cache and use more sophisticated cache replacement strategies. Future work includes multi-level partitioning designed to take advantage of the different levels of cache in a contemporary computer.

CRediT authorship contribution statement

Anuj Jain: Conceptualization, Methodology, Software, Validation, Investigation, Visualization, Writing - original draft, Writing - review & editing. **Sartaj Sahni:** Conceptualization, Methodology, Writing - original draft, Supervision, Project administration, Funding acquisition, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was funded, in part, by the National Science Foundation, under Contract No. 1748652.

References

[1] D. White, A survey of applications of Markov decision processes, J. Oper. Res. Soc. 44 (1993) 1073–1096.
[2] V. Rao, L. Thomas, Dynamic Models for Sales promotion policies, J. Oper. Res. Soc. 24 (3) (1973) 403–407.
[3] R. Sutton, A. Barto, Reinforcement Learning: An Introduction, second ed., The MIT Press Cambridge, Massachusetts London, 2012.
[4] P. Dai, D. Weld, J. Goldsmith, Topological value iteration algorithms, J. Artificial Intelligence Res. 42 (2011) 181–209.
[5] D. Wingate, K. Seppi, Efficient value iteration using partitioned models, in: Proceedings of the International Conference on Machine Learning and Applications, ICMLA 03, 2003, pp. 53–59.
[6] D. Wingate, K. Seppi, Prioritization methods for accelerating MDP solvers, J. Mach. Learn. Res. 6 (15) (2005) 851–881.
[7] A. Sidford, M. Wang, X. Wu, Y. Ye, Variance reduced value iteration and faster algorithms for solving Markov decision processes, 2017, CoRR, abs/1710.09988.
[8] S. Ruiz, B. Hernándezi, A Parallel solver for Markov decision process in crowd simulations, in: MICAI, 2015.
[9] A. Sapio, S.S. Bhattacharyya, M. Wolf, Efficient solving of Markov decision processes on GPUs using parallelized sparse matrices, in: DASIP, 2018, pp. 13–18.
[10] C. Caramanis, N. Dimitrov, D. Morton, Efficient algorithms for budget-constrained Markov decision processes, IEEE Trans. Automat. Control 59 (10) (2014).
[11] C. Zhao, S. Sahni, Cache and energy efficient algorithms for Nussinov RNA folding, BMC Bioinformatics 18 (15) (2017) 518.
[12] D. Bertsekas, J. Tsitsiklis, Neuro-Dynamic Programming, Athena Scientific, Belmont, Massachusetts, 1996.
[13] A. Jain, S. Sahni, Cache efficient value iteration, in: Symposium on Computers and Communications, ISCC, 2019.
[14] S. Nahar, S. Sahni, E. Shragowitz, Simulated annealing and combinatorial optimization, Int. J. Comput. Aided VLSI Des. 1 (1) (1989) 1–23.
[15] Perf tool: Profiler tool for Linux 2.6+ based systems, https://perf.wiki.kernel.org/index.php/Tutorial.