iterations. This section investigates reducing such wasteful backups by choosing an intelligent backup order. The key idea is to additionally define a *priority* of each state s, representing an estimate of how helpful it is to back up s. Higher-priority states are backed up earlier.

Algorithm 3.5: Prioritized Value Iteration

```
1  initialize V
2  initialize priority queue q
3  repeat
4  |  select state s' = q.pop()
5  |  compute V(s') using a Bellman backup at s'
6  |  foreach predecessor s of s', i.e., {s|∃a[T(s,a,s') > 0]} do
7  |  |  compute priority(s)
8  |  |  q.push(s, priority(s))
9  |  end
10 until termination;
11 return greedy policy π^V
```

When can we avoid backing up a state s? An obvious answer is: when none of the successors of s have had a change in value since the last backup — this would mean that backing up s wil not result in a change of its value. In other words, a change in the value of a successor is an indicator for changing the priority of a state.

We formalize this intuition in Algorithm 3.5 with a general Prioritized Value Iteration scheme (adapted from [246]). It implements an additional priority queue that maintains the priority for backing up each state. In each step, a highest-priority state is popped from the queue and a backup is performed on it (lines 4-5). The backup results in a potential change of priorities for all the predecessors of this state. The priority queue is updated to take into account the new priorities (lines 6-9). The process is repeated until termination (i.e., until $Res^V < \epsilon$).

Convergence: Prioritized VI is a special case of Asynchronous VI, so it converges under the same conditions — if all states are updated infinitely often. However, because of the prioritization scheme, we cannot guarantee that all states will get updated an infinite number of times; some states may get *starved* due to their low priorities. Thus, in general, we cannot prove the convergence of Prioritized VI. In fact, Li and Littman [151] provide examples of domains and priority metrics where Prioritized VI does not converge. We can, however, guarantee convergence to the optimal value function, if we simply interleave synchronous VI iterations within Prioritized VI. While this is an additional overhead, it mitigates the lack of theoretical guarantees. Moreover, we can prove convergence without this interleaving for specific priority metrics (such as prioritized sweeping).

Researchers have studied different versions of this algorithm by employing different priority metrics. We discuss these below.

3.5.1 PRIORITIZED SWEEPING

Probably the first and most popular algorithm to employ an intelligent backup order is Prioritized Sweeping (PS) [183]. PS estimates the expected change in the value of a state if a backup were to be performed on it now, and treats this as the priority of a state. Recall that $Res^V(s')$ denotes the change in the value of s' after its Bellman backup. Let s' was backed up with a change in value $Res^V(s')$ then PS changes the priority of all predecessors of s'. If the highest-probability transition between s and s' is p then the maximum residual expected in the backup of s is $p \times Res^V(s')$. This forms the priority of s after the backup of s':

$$\text{priority}_{PS}(s) \leftarrow \max\left\{\text{priority}_{PS}(s), \max_{a\in\mathcal{A}}\left\{\mathcal{T}(s,a,s')Res^V(s')\right\}\right\} \tag{3.5}$$

For the special case, where the state is a successor of itself, the outer max step is ignored: $\text{priority}(s') \leftarrow \max_{a\in\mathcal{A}}\{\mathcal{T}(s,a,s')Res^V(s')\}$.

Example: Continuing with example in Figure 3.2, say, we first back up all states in the order s_0 to s_4 (as in the first iteration of VI). So far, the only state that has changed value is s_4 (residual = 1.8). This increases the priorities of s_2 and s_3 to be 1.8, so one of them will be the next state to be picked up for updates. Note that the priority of s_0 will remain 0, since no change in its value is expected. □

Convergence: Even though PS does not update each state infinitely often, it is still guaranteed to converge under specific conditions [151].

Theorem 3.21 Prioritized Sweeping converges to the optimal value function in the limit, if the initial priority values are non-zero for all states $s \in \mathcal{S}$.

This restriction is intuitive. As an example, if all predecessors of goals have initial priorities zero (s_4 in our example), a goal's base values will not be propagated to them and hence, to no other states. This may result in converging to a suboptimal value function.

The practical implementations of PS for SSP MDPs make two additional optimizations. First, they initialize the priority queue to contain only the goal states. Thus, the states start to get backed up in the reverse order of their distance from goals. This is useful, since values flow from the goals to the other states. Second, for termination, they update the priorities only if $\text{priority}_{PS}(s) > \epsilon$. Thus, the algorithm does not waste time in backing up states where the residual is expected to be less than ϵ.

Generalized Prioritized Sweeping: Generalized PS [4] suggests two priority metrics. The first is a modification of PS where priorities are not maxed with the prior priority, rather, added as follows:

$$\text{priority}_{GPS1}(s) \leftarrow \text{priority}_{GPS1}(s) + \max_{a\in\mathcal{A}}\left\{\mathcal{T}(s,a,s')Res^V(s')\right\} \tag{3.6}$$

This makes the priority of a state an *overestimate* of its residual. The original intent of GPS1's authors was that this would avoid starvation, since states will have a higher priority than their residual. But, as was later shown, this scheme can potentially starve other states [151].

The second metric (GPS2) computes the actual residual of a state as its priority:

$$\begin{aligned} \text{priority}_{GPS2}(s) &\leftarrow Res^V(s) \\ &= \left| \sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s') \left[C(s,a,s') + V(s') \right] - V(s) \right| \end{aligned} \tag{3.7}$$

The implementation of the GPS2 priority scheme differs from our pseudocode for Prioritized VI in one significant way — instead of estimating the residual, it uses its *exact* value as the priority. In other words, one needs to perform the backup to even compute the priority function. The effect is that value propagation happens at the time a state is *pushed* in the queue (as opposed to when it is popped, in the case of PS). This particular metric does not require interleaved synchronous VI iterations for convergence [151].

3.5.2 IMPROVED PRIORITIZED SWEEPING

PS successfully incorporates the intuition that if the value of a successor, $V(s')$, has changed, then the values of the predecessor s may have changed also, so its backup needs to be scheduled. While this insight is accurate and useful, it is also somewhat myopic. At the global level, we wish to iterate these values to their fixed-points. If $V(s')$ is still in flux, then any further change in it will require additional propagations to s.

For example, Figure 3.3 shows that the values of s_3 and s_4 converge slowly due to their interdependencies. And each iteration propagates their intermediate values to $s_0 - s_2$. These are essentially wasted computations. We could save a lot of computation if we first allowed s_3 and s_4 to converge completely and then propagated their values to $s_0 - s_2$ just once. Of course, since we are dealing with cyclic graphs, this may not always be possible. However, we could at least incorporate this insight into our prioritization scheme.

In SSP MDPs, values propagate from goals backwards to other states. To include proximity to a goal as a criterion in prioritization, Improved Prioritized Sweeping[2] [175] defines the following priority metric (only defined for non-goals, $V(s) > 0$):

$$\text{priority}_{IPS}(s) \leftarrow \frac{Res^V(s)}{V(s)} \tag{3.8}$$

The division by $V(s)$ trades off residual and proximity to a goal. Since the states closer to a goal will have smaller $V(s)$, thus, their priority will be higher than states farther away. Slowly, as their residual reduces, states farther away become more important and they get backed up. Note that

[2]Their original paper did not perform Bellman backups, and presented a different expansion-based algorithm. Their metric in the context of Prioritized VI was discussed first in [60].

the implementation of this algorithm is similar to that of the second variant of Generalized PS – the states are backed up when they are pushed in the queue rather than when they are popped.

3.5.3 FOCUSED DYNAMIC PROGRAMMING

In the special case when the knowledge of a start state s_0 is available (discussed in detail in Chapter 4), the prioritization scheme can also take into account the relevance of a state with respect to the solution from the start state. *Focused Dynamic Programming* [87; 88] defines the priority metric $\text{priority}_{FDP}(s) \leftarrow h(s) + V(s)$. Here, $h(s)$ is a lower bound on the expected cost of reaching s from s_0. Note that for this algorithm, lower priority values are backed up first.

3.5.4 BACKWARD VALUE ITERATION

In all these prioritized algorithms the existence of a priority queue is a given. However, sometimes the overhead of a priority queue can be significant. Especially, for large domains, where states have a bounded number of successors, a Bellman backup takes constant time but each priority queue update may take $O(log|\mathcal{S}|)$ time, which can be very costly. Wingate & Seppi [246] provide an example of a domain where Generalized PS performs half as many backups as VI but is 5-6 times slower, with the bulk of the running time attributed to priority queue operations.

In response, the Backward Value Iteration (BVI) algorithm [60] suggests performing backups simply in the reverse order of the distance from a goal using a FIFO queue (as opposed to a priority queue). Another optimization, instead of backing up all states in this fashion, backs up only the states that are in the greedy subgraph starting from a goal state (backward).

Algorithm 3.6 elaborates on the BVI algorithm. In each iteration, the queue starts with the goal states and incrementally performs backups in the FIFO order. A state is never reinserted into the queue in an iteration (line 11). This guarantees termination of a single iteration. Finally, note line 10. It suggests backing up only the states that are in the current greedy policy. This is in contrast with other PS algorithms that back up a state if they are a predecessor of the current backed up state, irrespective of whether the action is currently greedy or not. It is not clear how important this modification is for the performance of BVI, though similar ideas have been effective in other algorithms, e.g., ILAO* (discussed in Section 4.3.2).

No convergence results are known for BVI, though our simple trick of interleaving VI iterations will guarantee convergence. In practice, BVI saves on the overhead of priority queue implementation, and in many domains it also performs far fewer backups than other PS algorithms. The results show that it is often more efficient than other PS approaches [60].

3.5.5 A COMPARISON OF PRIORITIZATION ALGORITHMS

Which prioritization algorithm to choose? It is difficult to answer this question, since different algorithms have significantly different performance profiles and they all work very well for some domains.

Algorithm 3.6: Backward Value Iteration

```
1  initialize V arbitrarily
2  repeat
3      visited = ∅
4      ∀g ∈ G, q.push(g)
5      while q ≠ ∅ do
6          s' = q.pop()
7          visited.insert(s')
8          compute V(s') using a Bellman backup at s'
9          store π(s') = π^V(s') //greedy action at s'
10         foreach {s|T(s, π(s), s') > 0} do
11             if s not in visited then
12                 q.push(s)
13             end
14         end
15     end
16 until termination;
17 return greedy policy π^V
```

Non-prioritized (synchronous) VI provides very good backup orders in cases where the values of states are highly interdependent. For example, for a fully connected subgraph, one cannot possibly do much better than VI. On such domains, prioritized methods add only computational overhead and no value. On the other hand, if a domain has highly sequential dependencies, then VI with random state orders falls severely short.

PS and Generalized PS obtain much better performance for domains with sequential dependencies, such as acyclic graphs or graphs with long loops. They work great for avoiding useless backups but tend to be myopic, since they do not capture the insight that one can save on propagating all intermediate information to farther-away states by first letting the values closer to goals stabilize.

Improved PS suggests one specific way to trade off between proximity to a goal and residual as means of prioritization. However, this particular tradeoff may work really well for some domains, and may give rather suboptimal backup orders for others.

Finally, BVI is a priority-queue-free algorithm, which completely shuns any residual-based prioritization and focuses entirely on ordering based on distance from a goal (in the greedy subgraph). It is shown to perform significantly better than other PS algorithms for several domains, but it may fail for domains with large fan-ins (i.e., with states with many predecessors). An extreme case is SSP MDPs compiled from infinite-horizon discounted-reward problems (refer to the transformation discussed in Section 2.4), where all states have a one-step transition to a goal.

3.6 PARTITIONED VALUE ITERATION

We now focus our attention on another important idea for managing VI computations — the idea of partitioning the state space. We illustrate this again with the example of Figure 3.2. What is the optimal backup order for this MDP? We note that states s_3 and s_4 are mutually dependent, but do not depend on values of $s_0 - s_2$. We will need several iterations for the values of s_3 and s_4 to converge. Once that happens, the values can be backed up to s_1 and s_2, and finally to s_0. This example clearly illustrates that we need to stabilize mutual co-dependencies before attempting to propagate the values further.

We can formalize this intuition with the notion of Partitioned Value Iteration [246] – we create a partitioning of states (in our example, $\{\{s_0\}, \{s_1, s_2\}, \{s_3, s_4\}\}$ makes the best partitioning for information flow) and perform several backups within a partition before focusing attention on states in other partitions.

Additionally, we can also incorporate a prioritization scheme for choosing an intelligent *partition order*. In our example, partitions need to be considered in the order from right to left. All the intuitions from the previous section can be adapted for the partitioned case. However, instead of successor/predecessor states, the notion of successor/predecessor partitions is more relevant here. For partition $\mathfrak{p}'$, we can define another partition $\mathfrak{p}$ to be a *predecessor partition* if some state in $\mathfrak{p}$ is a predecessor of a state in $\mathfrak{p}'$. Formally,

$$\text{PredecessorPartition}(\mathfrak{p}') = \{\mathfrak{p} | \exists s \in \mathfrak{p}, \exists s' \in \mathfrak{p}', \exists a \in \mathcal{A}\ s.t.\ \mathcal{T}(s, a, s') > 0\} \tag{3.9}$$

Algorithm 3.7 provides pseudocode for a general computation scheme for Partitioned VI. If line 6 only executes one backup per state and the prioritization scheme is round-robin, then Partitioned VI reduces to VI. If all partitions are of size 1 then the algorithm, depending on the priority, reduces to various PS algorithms. Another extreme case is running each partition till convergence before moving to another partition. This version is also known as *Partitioned Efficient Value Iterator (P-EVA)* [243].

Which algorithm should one use within a partition (line 6)? Essentially any algorithm can be used, from VI, PI to any of the prioritized versions. The convergence of Partitioned VI follows the same principles as before. If we can guarantee that the residual of each state is within ϵ, then the algorithm has converged and can be terminated. This can be achieved by interleaving VI sweeps or ensuring any other way that no state/partition starves.

More importantly, how do we construct a partitioning? Different approaches have been suggested in the literature, but only a few have been tried. Intuitively, the partitioning needs to capture the underlying co-dependency between the states – states whose values are mutually dependent need to be in the same partition. If the states have specific structure, it can be exploited to generate partitions. Say, if the states have associated spatial location, then nearby states can be in the same partition. Domain-independent ways to partition states could employ k-way graph partitioning or

Algorithm 3.7: Partitioned Value Iteration

```
initialize V arbitrarily
construct a partitioning of states 𝔓 = {𝔭_i}
(optional) initialize priorities for each 𝔭_i
repeat
    select a partition 𝔭′
    perform (potentially several) backups for all states in 𝔭′
    (optional) update priorities for all predecessor partitions of 𝔭′
until termination;
return greedy policy π^V
```

clustering algorithms [3; 69; 184; 186]. We now discuss a special case of Partitioned VI that employs a specific partitioning approach based on strongly-connected components.

3.6.1 TOPOLOGICAL VALUE ITERATION

Topological Value Iteration (TVI) [59; 66] uses an MDP's graphical structure to compute a partitioning of the state space. It first observes that there is an easy way to construct an optimal backup order in the case of acyclic MDPs [21].

Theorem 3.22 *Optimal Backup Order for Acyclic MDPs.* If an MDP is acyclic, then there exists an optimal backup order. By applying the optimal order, the optimal value function can be found with each state needing only one backup.

The reader may draw parallels with the policy computation example from Figure 3.1(a). In the case of an acyclic graph, we can propagate information from goal states backward and return an optimal solution in a single pass. Of course, the interesting case is that of cyclic MDPs, where such an order does not exist. But we may adapt the same insights and look for an *acyclic partitioning*. If we could find a partitioning where all partitions are linked to each other in an acyclic graph then we could back up partitions in the optimal order and run VI within a partition to convergence.

The key property of this partitioning is that the partitions and links between them form an acyclic graph. That is, it is impossible to go back from a state in a succeeding partition to a state in the preceding partition. Thus, all states within a partition form a *strongly connected component (SCC)*: we can move back and forth between all states in an SCC. However, this property does not hold true across different SCCs.

The computation of SCCs is well studied in the algorithm literature. TVI first converts the MDP graph into a deterministic causal graph, which has a directed edge between two states s and s', if there is a non-zero probability transition between the two: $\exists a \in \mathcal{A}[\mathcal{T}(s, a, s') > 0]$. TVI then applies an algorithm (e.g., the Kosaraju-Sharir algorithm, which runs linear in the size of the graph [55]) to compute a set of SCCs on this graph. These are used as partitions for Partitioned VI. The reverse topological sort on the partitions generates an *optimal* priority order to backup

these partitions in the sense that a partition is backed up after all its successors have been run to ϵ-consistency.

Theorem 3.23 Topological Value Iteration for an SSP MDP is guaranteed to terminate with an ϵ-consistent value function, as long as $\epsilon > 0$.

Example: In the example MDP from Figure 3.2, an SCC algorithm will identify the partitions to be $\mathfrak{p}_0 = \{s_0\}, \mathfrak{p}_1 = \{s_1, s_2\}, \mathfrak{p}_2 = \{s_3, s_4\}\mathfrak{p}_3 = \{g\}$. The reverse topological sort on the partitions will return the backup order as $\mathfrak{p}_3, \mathfrak{p}_2, \mathfrak{p}_1, \mathfrak{p}_0$. Since $\mathfrak{p}_3$ only has the goal state, it does not need backups. TVI will run VI for all states within $\mathfrak{p}_2$ next. Once V_n for s_3 and s_4 converge, they will be propagated to $\mathfrak{p}_1$. The values from $\mathfrak{p}_1$ will propagate to $\mathfrak{p}_0$ to compute $V^*(s_0)$. □

TVI offers large speedups when a good SCC-based partitioning exists, i.e., each partition is not too large. However, often, such a partitioning may not exist. For example, any reversible domain (each state is reachable from all other states) has only one SCC. For such problems, TVI reduces to VI with an additional overhead of running the SCC algorithm, and is not effective. An extension of TVI, known as *Focused Topological Value Iteration* prunes suboptimal actions and attains good partitioning in a broad class of domains. This algorithm is discussed in Section 4.5.2.

Partitioned VI naturally combines with a few other optimizations. We discuss a few below when describing external-memory algorithms, cache-efficient algorithms, and parallel algorithms.

3.6.2 EXTERNAL MEMORY/CACHE EFFICIENT ALGORITHMS

VI-based algorithms require $O(|\mathcal{S}|)$ memory to store current value functions and policies. In problems where the state space is too large to fit in memory, disk-based algorithms can be developed. However, since disk I/O is a bottleneck, value iteration or general prioritized algorithms may not perform as well. This is because these require random access of state values (or a large number of sweeps over the state space), which are I/O-inefficient operations.

Partitioned VI fits nicely in the external memory paradigm, since it is able to perform extensive value propagation within a partition before focusing attention on the next partition, which may reside on the disk. Thus, Partitioned VI may terminate with much fewer I/O operations.

Partitioned External Memory Value Iteration (PEMVI) [61] partitions the state space so that all information relevant to backing up each partition (transition function of this partition and values of current and all successor partitions) fits into main memory. It searches for such a partitioning via domain analysis.[3] The details of this work are out of the scope of this book. The reader may refer to the original paper for details [62].

An interesting detail concerns PEMVI's backup orders. There are two competing backup orders — one that minimizes the I/O operations and one that maximizes the information flow (reduces number of backups). The experiments show that the two are not much different and have similar running times.

[3]The partitioning algorithm investigates the special case when the domains are represented in Probabilistic PDDL [253] representation.

Partitioning is not the only way to implement external-memory MDP algorithms. *External Memory Value Iteration* is an alternative algorithm that uses clever sorting ideas to optimize disk I/Os [82]. In practice, PEMVI with good partitioning performs better than EMVI [61].

Closely associated with the I/O performance of an external-memory algorithm is the cache efficiency of the internal memory algorithm. To our knowledge, there is only one paper that studies cache performance of the VI-based algorithms [244]. It finds that Partitioned VI, in particular, the P-EVA algorithm (with specific priority functions), obtains a much better cache performance. This is not a surprising result, since, by partitioning and running backups till convergence on the current partition, the algorithm naturally accesses the memory with better locality.

We believe that more research on cache efficiency of MDP algorithms is desirable and could lead to substantial payoffs — similar to the literature in the algorithms community (e.g., [147]), one may gain huge speedups due to better cache performance of the algorithms.

3.6.3 PARALLELIZATION OF VALUE ITERATION

VI algorithms are easily parallelizable, since different backups can be performed in parallel. An efficient parallel algorithm is P3VI, which stands for *Partitioned, Prioritized, Parallel Value Iterator*. Breaking the state space into partitions naturally allows for efficient parallelized implementations — different partitions are backed up in parallel.

The actual algorithm design is more challenging than it sounds, since it requires us to construct a strategy to allocate processors to a partition, and also manage the messages between the processors when the value function of a partition has changed, which may result in reprioritizing the partitions. We refer the reader to the original paper [245] for those details.

The stopping condition is straightforward. P3VI terminates when all processors report a residual less than ϵ. Unsurprisingly, the authors report significant speedups compared to naive multi-processor versions of the VI algorithm.

3.7 LINEAR PROGRAMMING FORMULATION

While iterative solution algorithms are the mainstay of this book, no discussion about MDP algorithms is complete without a mention of an alternative solution approach that is based on linear programming. The set of Bellman equations for an SSP MDP can also be solved using the following LP formulation [78] ($\alpha(s) > 0, \forall s$):

$$
\begin{aligned}
\text{Variables} \quad & V^*(s) \quad \forall s \in \mathcal{S} \\
\text{Maximize} \quad & \sum_{s \in \mathcal{S}} \alpha(s) V^*(s) \\
\text{Constraints} \quad & V^*(s) = 0 \quad \text{if } s \in \mathcal{G} \\
& V^*(s) \leq \sum_{s' \in \mathcal{S}} [\mathcal{C}(s,a,s') + \mathcal{T}(s,a,s') V^*(s')] \quad \forall s \in \mathcal{S} \setminus \mathcal{G}, a \in \mathcal{A}
\end{aligned}
$$