Numerical Methods for the Hamilton-Jacobi-Bellman Equation in Optimal Control

Konstantinos Christopher Tsiolis
Department of Mathematics and Statistics
McGill University
kc.tsiolis@mail.mcgill.ca

Abstract

We provide a treatment of the dynamic programming approach to optimal control problems, an approach to which the Hamilton-Jacobi-Bellman partial differential equation is central. Using the minimum time to target problem as our guiding example, we show how the fast sweeping method for the eikonal equation can be used as a solution method for optimal control. In addition, we relate the use of dynamic programming in optimal control to its use in reinforcement learning. For a similarly posed minimum time to target problem, we view the Bellman equation and value iteration as adaptations of the Hamilton-Jacobi-Bellman equation and fast sweeping to the discrete-time setting.

1 Introduction

At a high level, the optimal control problem is one of influencing the trajectory of an object so as to minimize some cost (or maximize some return). [2] provides a comprehensive early history of optimal control, explaining how it developed in the mid-20th century out of the older calculus of variations. It was conceived especially for problems in aerospace and robotics. Of special interest in the aerospace context was finding optimal (in the sense of minimum time) flight paths to reach a certain altitude given constraints on fuel.

The constraints of an optimal control problem are typically described by a system of ordinary differential equations (ODEs) where time is the independent variable. These equations describe the evolution of the state of the system (e.g., position, amount of fuel remaining). These dynamics depend on the *control* (e.g., thrust), which is determined by the "user". As the name suggests, we only have influence over the control.

Richard Bellman proposed the dynamic programming method [1] in the 1950s as a means of addressing the optimal control problem. His work showed that the optimal control can be found by solving a partial differential equation, the *Hamilton-Jacobi-Bellman* equation. The solution to this equation, referred to as the *value function*, assigns a value to each state. The optimal control is then the one that navigates towards states with high value. These notions will be formalized in Section 2.

In our discussions of numerical methods and our experiments with continuous-time optimal control, we focus on the minimum-time problem. The goal is to reach a target set from a point in the state space in a minimal amount of time. If the speed of the object depends only on the current position (and not on the current time), we show in Section 4.1 that the Hamilton-Jacobi-Bellman equation reduces to the *eikonal equation*. In Section 5.1, we introduce the fast sweeping method [14], an efficient numerical method for solving the eikonal equation. We then apply the fast sweeping method

to a simple minimum-time problem where the target set is the closed unit ball in $\mathbb{R}^2$. We are able to approximate the true value function to two digits of accuracy with 1024 grid points (c.f. Section 6.1).

Throughout this paper, we will draw connections between optimal control and the more recently developed field of reinforcement learning. The latter has seen a surge in popularity due to the success of deep learning methods [7, 11]. In the simplest case, reinforcement learning can be viewed as the application of optimal control to discrete-time problems. At a given time step, we find ourselves in a given state and select an action, which then takes us to a new state at the next time step and induces a reward. The goal is to maximize the cumulative reward, which is referred to as the *return*. In Section 3, we show that there is an analog to the Hamilton-Jacobi-Bellman equation in this setting, which is simply called the *Bellman equation*. In the tabular reinforcement learning setting (where the state space is finite and reasonably small), the Bellman equation can be used to develop a fixed-point iteration. This *value iteration* provably converges to the optimal value function, as we will see in Section 5.2. We experiment with value iteration on the tabular gridworld environment in Section 6.2.

It should be noted that there is a key limitation to the parallels we draw between optimal control and reinforcement learning in this paper. Even more important than the issue that tabular reinforcement learning only captures a small subset of reinforcement learning problems of interest [12] is the issue that we do not have access to the dynamics of the system in most reinforcement learning problems. That is, in most cases, we do not know how the system will respond when we take an action. It is in this context that reinforcement learning begins to diverge from optimal control as we pose it here. Reinforcement learning agents must – either implicitly or explicitly – learn a model of the system's dynamics in order to select optimal actions.

2 Optimal Control with Dynamic Programming

In this section, we formally introduce the optimal control setting, from which the Hamilton-Jacobi-Bellman equation will arise naturally. We draw on the book [6] and the introductory notes [8] and [5].

To begin, consider the dynamics described by the following ordinary differential equation (ODE):

$$\begin{cases} x'(t) = f\big(x(t), \alpha(t)\big), & t > 0 \\ x(0) = x_0. \end{cases} \tag{1}$$

Here, we think of $x(\cdot) : [0, \infty) \to \mathbb{R}^n$ as describing the position of a physical entity over time with x_0 as the starting point. $\alpha(\cdot) : [0, \infty) \to A$ is the *control* which exerts an influence on the trajectory $x(\cdot)$. For example, we can imagine that $x(\cdot)$ represents the position of a car and $\alpha(\cdot)$ captures the gas and brake pedal. At each time step t, the control selects an action $\alpha(t) \in A$, where $A \subseteq \mathbb{R}^m$ denotes the set of admissible actions. Following [6] and [5], we take the set of admissible controls to be

$$\mathcal{A} = \{\alpha : [0, \infty) \to A : \alpha(\cdot) \text{ is measurable}\}.$$

Then, the objective in an optimal control problem is to select the control $\alpha(\cdot)$ which maximizes the *return*

$$G_{x,t}[\alpha(\cdot)] := \int_t^T r\big(x(t), \alpha(t)\big)\, dt + g\big(x(T)\big), \tag{2}$$

where $x(t) = x$, T is some terminal time (though we can in principle also consider infinite-horizon problems), r is the *running reward*, and g is the *terminal reward*.

Remark 1. *In this paper, we use the standard terminology of reinforcement learning in describing optimal control problems, so as to emphasize the connection between the two. In [5], G is referred to as the "payoff". In [6], optimal control is equivalently formulated as minimizing a "cost" function.*

When solving optimal control problems with dynamic programming, the main object of interest is the *value function*

$$u(x, t) = \sup_{\alpha(\cdot) \in \mathcal{A}} G_{x,t}\big[\alpha(\cdot)\big] \tag{3}$$

for $x \in \mathbb{R}^n$ and $0 \leq t \leq T$. This function captures the largest achievable return given the set of admissible controls.

A major result is that the value function arises as the solution to the Hamilton-Jacobi-Bellman equation, which we introduce in the theorem below. We present the theorem as it is stated in [5].

Theorem 1 (Hamilton-Jacobi-Bellman Equation)**.** *Assume that the value function u is continuously differentiable. Then u is the solution of the Hamilton-Jacobi-Bellman equation*

$$\begin{cases} u_t(x,t) + \max_{a \in A} \left\{ \langle f(x,a), \nabla_x u(x,t) \rangle + r(x,a) \right\} = 0, & x \in \mathbb{R}^n, 0 \leq t < T \\ u(x,T) = g(x), & x \in \mathbb{R}^n. \end{cases} \tag{4}$$

Remark 2. *The above theorem assumes that the value function is continuously differentiable. This is not necessarily true. To handle the non-C^1 case, it is necessary to consider weak solutions of the Hamilton-Jacobi-Bellman equation. In this context, it can be shown that the value function corresponds to the unique viscosity solution of the Hamilton-Jacobi-Bellman equation [3, 6].*

Proof. We follow the proof from [5].

Let $t \geq 0$, $h > 0$, and $a \in A$. Suppose $\alpha(\cdot) = a$ for all $t \leq s \leq t + h$ and some fixed $a \in A$. Then, from the definition of return and the value function,

$$u(x,t) \geq \int_t^{t+h} r\big(x(s), a\big)\, ds + u\big(x(t+h), t+h\big).$$

Rearranging and dividing by h, we have (by the Chain Rule),

$$\frac{u\big(x(t+h), t+h\big) - u(x,t)}{h} + \frac{1}{h} \int_t^{t+h} r\big(x(s), a\big)\, ds \leq 0.$$

Letting $h \to 0$, we have

$$u_t(x,t) + \langle \nabla_x u\big(x,t\big), x'(t) \rangle + r\big(x,a\big) \leq 0.$$

We can use Equation 1 to write the derivative x' in terms of f:

$$u_t(x,t) + \langle \nabla_x u(x,t), f(x,a) \rangle + r(x,a) \leq 0.$$

Since the above holds for any $a \in A$, we have

$$u_t(x,t) + \max_{a \in A} \left\{ \langle f(x,a), \nabla_x u(x,t) \rangle + r(x,a) \right\} \leq 0.$$

It remains to show that the above is in fact an equality. Suppose $\alpha^*(\cdot)$ is the optimal control and $x^*(\cdot)$ is the associated trajectory. Then, we can follow the same steps as above. We can decompose the value function as

$$u(x,t) = \int_t^{t+h} r\big(x^*(s), \alpha^*(s)\big)\, ds + u\big(x^*(t+h), t+h\big).$$

Rearranging and dividing by h, we have

$$\frac{u\big(x^*(t+h), t+h\big) - u(x,t)}{h} + \frac{1}{h} \int_t^{t+h} r\big(x^*(s), \alpha^*(s)\big)\, ds = 0.$$

Let $h \to 0$ and suppose $\alpha^*(t) = a^* \in A$. Then

$$u_t(x,t) + \langle \nabla_x u(x,t), x^{*'}(t) \rangle + r(x,a^*) = 0.$$

Once again using Equation 1, we conclude

$$u_t(x,t) + \langle f(x,a^*), \nabla_x u(x,t) \rangle + r(x,a^*) = 0$$

for some $a^* \in A$. □

The above proof is illustrative of the *Dynamic Programming Principle* introduced by Richard Bellman [1]. Using the value function to quantify the return we hope to obtain, we can decompose this return along the trajectory as the return over a small period of time $[t, t+h]$ plus the value function evaluated at the end of that period. In this way, we have a recursive formulation for the value function.

In addition to providing the maximum return, the value function also indicates which action should be selected – thus allowing us to solve the optimal control problem. The following theorem, stated and proven in [5], illustrates how to select an optimal control from a value function.

Theorem 2. *Given the value function u for an optimal control problem, the control defined by*

$$\alpha^*(t) = \arg\max_{a \in A} \big[\langle f(x(t), a), \nabla_x u(x(t), t)\rangle + r(x(t), a)\big] \tag{5}$$

is optimal.

Proof. Let $x^*(\cdot)$ be the trajectory associated to $\alpha^*(\cdot)$.

$$\begin{aligned}
G_{x,t}[\alpha^*(\cdot)] &= \int_t^T r\big(x^*(s), \alpha^*(s)\big)\, ds + g\big(x^*(T)\big) \\
&= \int_t^T \big(- u_t\big(x^*(s), s\big) - \langle f\big(x^*(s), \alpha^*(s)\big), \nabla_x u\big(x^*(s), s\big)\rangle\big)\, ds + g\big(x^*(T)\big) \\
&= -\int_t^T u_t\big(x^*(s), s\big) + \langle \nabla_x u\big(x^*(s), s\big), x^{*'}(s)\rangle\, ds + g\big(x^*(T)\big) \\
&= -\int_t^T \frac{d}{ds}\big[u\big(x^*(s), s\big)\big]\, ds + g\big(x^*(T)\big) \\
&= -u(x^*(T), T) + u(x, t) + g(x^*(T)) \\
&= u(x, t) \\
&= \sup_{\alpha(\cdot) \in \mathcal{A}} G_{x,t}[\alpha(\cdot)],
\end{aligned}$$

where the second line follows from the Hamilton-Jacobi-Bellman equation, the third line follows from the ODE, and the fourth line follows from Chain Rule. □

3 Reinforcement Learning

Reinforcement learning can be viewed as a discrete-time version of optimal control. Here, we replace our controlled ODE model (1) with a *Markov Decision Process* (MDP). The latter is a discrete-time stochastic process denoted by a tuple (S, A, R, P, P_0), where

- S is the state space;
- A is the action space;
- R is the set of possible rewards;
- $P : S \times A \times S \times R \to [0, 1]$ is the transition probability function;
- $P_0 : S \to [0, 1]$ is a probability function over states (the initial state distribution).

In an MDP, we begin in some initial state $S_0 \sim P_0$. At each time step $t \geq 0$, we select an action A_t, which triggers a transition to a new state S_{t+1} and incurs a reward R_{t+1} based on $P(\cdot|S_t, A_t)$. Actions are selected based on a learned *policy* $\pi(\cdot|s) : A \to [0, 1]$ which assigns a probability to each action given the current state.

The goal of a reinforcement learning problem is to find a policy which maximizes the return

$$G = \sum_{t=0}^{T} \gamma^t R_{t+1}, \tag{6}$$

where $0 < \gamma \leq 1$ is a *discount factor* and T is an (optional) terminal time.[1]

Just as the value function in optimal control aids us in accomplishing the task of maximizing return, we have an analogous value function in reinforcement learning. We follow the definitions and notation from [12]. Given a policy π, we have the *state value function*

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right]. \tag{7}$$

for all $s \in S$. This gives us the expected return incurred following policy π given that we start in state s.

For simplicity, assume that S, A, R are finite. Here, we have an analog to the Hamilton-Jacobi-Bellman equation, which is simply referred to as the *Bellman equation*. This gives us a recursive formulation of the state value function.

Theorem 3 (Bellman Equation)**.** *Given a policy π, the state value function $v_\pi(s)$ can be expressed as*

$$v_\pi(s) = \pi(a|s) \sum_{s' \in S, r \in R} p(S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a)\big[r + \gamma v_\pi(s')\big]. \tag{8}$$

To simplify the notation for the proof and later discussions, we will henceforth write transition probabilities as $p(s', r|s, a)$.

Proof. We follow the proof in [12]. We have

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \middle| S_t = s\right] \\
&= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a)\big[r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s']\big] \\
&= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a)\big[r + v_\pi(s')\big].
\end{aligned}$$

□

Since we are interested in the policy which maximizes the return, we arrive at a more direct analog to our value function from optimal control,

$$v_*(s) = \max_\pi v_\pi(s), \tag{9}$$

which is referred to as the *optimal state-value function* [12].

This leads us the following key theorem.

Theorem 4 (Bellman Optimality Equation)**.** *The optimal state-value function v_* satisfies*

$$v_*(s) = \max_a \sum_{s', r} p(s', r|s, a)\big[r + \gamma v_*(s')\big]. \tag{10}$$

[1] If we take $T = \infty$, then we must have $\gamma < 1$ in order to have finite return.

Proof. We once again follow the proof in [12]. Let π_* be an optimal policy, i.e., $v_{\pi_*} = v_*$. Then,

$$\begin{aligned} v_*(s) &= \mathbb{E}_{\pi_*}[G_t|S_t = s] \\ &= \max_a \mathbb{E}_{\pi_*}[G_t|S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}\big[R_{t+1} + \gamma G_{t+1}\big|S_t = s, A_t = a\big] \\ &= \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma \mathbb{E}_{\pi_*}[G_{t+1}|S_{t+1} = s']\big] \\ &= \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma v_*(s')\big]. \end{aligned}$$

□

4 Examples of Optimal Control Problems

In this section, we describe three simple optimal control problems and their associated Hamilton-Jacobi-Bellman equation (or just Bellman equation in the discrete-time case). This will naturally lead us to a discussion of numerical methods for finding the value function.

4.1 Distance to a Set

Consider the setting of an object travelling at nonzero speed $F(x)$ (i.e., the speed can depend on the position, but not on time) towards a bounded nonempty target set Ω. The goal is to reach Ω – which is equivalent to reaching $\Gamma := \partial\Omega$ – in the least time. Hence, we have the following dynamics, which are independent of time:

$$x' = Fa, \qquad ||a||_2 = 1.$$

That is, the control selects the direction in which to move. Since we take the perspective of maximizing return, we take the return to be the negative of the time τ to reach Γ from a point $x \in \mathbb{R}^n$ by taking the actions prescribed by α.

$$G_x[\alpha(\cdot)] = -\int_0^\tau 1\, dt = -\tau,$$

Notice that the return does not depend on the current time t.

The value function $u(x)$ is then simply the negative of the minimum time required to reach Γ from a point $x \in \mathbb{R}^n$. If we are travelling at constant unit speed, the value function is also equal to the negative of the distance from the point x to the set Ω.

Due to time independence, we have $u_t = 0$. Therefore, the Hamilton-Jacobi-Bellman equation is

$$\max_{||a||_2=1} \big\{\langle Fa, \nabla_x u\rangle - 1\big\} = 0,$$

which, when paired with the terminal condition, becomes

$$\begin{cases} ||\nabla_x u|| = 1/F \\ u(x) = 0, \qquad\qquad x \in \Gamma. \end{cases} \tag{11}$$

This is called the *eikonal equation*.

4.2 Rocket Railroad Car

The rocket railroad car problem is a classic toy problem for optimal control. We present the formulation from [5].

Suppose that we have a railroad car with rocket propellers attached to each end. We can control the rockets so as to provide a thrust in the positive or negative direction (suppose the magnitude of the thrust is bounded by 1). The goal of the problem is to reach the origin at a velocity of zero given that we start at some other point with a velocity of zero.

For this problem, the state $x(\cdot) = [x_1(\cdot), x_2(\cdot)]^T$ is two-dimensional. $x_1(\cdot)$ captures the position along the railroad track and $x_2(\cdot)$ captures the velocity. The action space is one-dimensional, with actions corresponding to acceleration. Hence, we have the dynamics

$$\begin{cases} x_1'(t) = x_2(t), & t \geq 0 \\ x_2'(t) = \alpha(t), & t \geq 0 \\ x_1(0) = x_1^0 \\ x_2(0) = x_2^0. \end{cases}$$

We can write this more compactly as

$$\begin{cases} x'(t) = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \alpha(t), & t \geq 0 \\ x(0) = x^0. \end{cases} \tag{12}$$

Similarly to the distance problem in the previous subsection, the return is the negative time to reach the origin

$$G_x[\alpha(\cdot)] = -\int_0^T 1 \, dt = -\tau.$$

The Hamilton-Jacobi-Bellman equation is then

$$\max_{|a| \leq 1} \{x_2 u_{x_1} + a u_{x_2} - 1\} = 0, \tag{13}$$

which, when paired with the terminal condition, becomes

$$\begin{cases} x_2 u_{x_1} + |u_{x_2}| = 1 \\ u(0,0) = 0. \end{cases} \tag{14}$$

From (13), we notice that $\alpha^*(x_1, x_2) = \text{sgn}(u_{x_2})$.

[5] shows that the solution to (14) is

$$u(x_1, x_2) = \begin{cases} -x_2 - 2\big(x_1 + \frac{1}{2}x_2^2\big)^{\frac{1}{2}}, & x_1 \geq -\frac{1}{2}x_2|x_2| \\ x_2 - 2\big(-x_1 + \frac{1}{2}x_2^2\big)^{\frac{1}{2}}, & x_1 < -\frac{1}{2}x_2|x_2|. \end{cases} \tag{15}$$

Hence, when $x_1 \geq -\frac{1}{2}x_2|x_2|$, we have

$$u_{x_2} = -1 - \left(x_1 + \frac{1}{2}x_2^2\right)^{-\frac{1}{2}} x_2 \tag{16}$$

and when $x_1 < -\frac{1}{2}x_2|x_2|$, we have

$$u_{x_2} = 1 - \left(-x_1 + \frac{1}{2}x_2^2\right)^{-\frac{1}{2}} x_2. \tag{17}$$

We visualize the exact value function and the optimal control in Figures 1 and 2 respectively. The optimal control reflects the intuition one might have about the problem. To see this, suppose that we begin on the right-hand side of the origin. Then, according to the optimal control, it is best to maximally accelerate in the direction of the origin ($a = -1$) until we reach a certain proximity. Then, maximally decelerate / brake ($a = 1$) so as to ensure that we reach the origin at zero velocity.

4.3 Gridworld

Gridworld is a simple example of a reinforcement learning problem. Suppose that we have a d-dimensional grid where all sides have length n. We index points in the grid as vectors $[v_1, \ldots, v_d]$, where $v_i \in \{0, 1, \ldots, n-1\}$ for $1 \leq i \leq d$. Suppose that we have a reward of 1 at the "corner" state

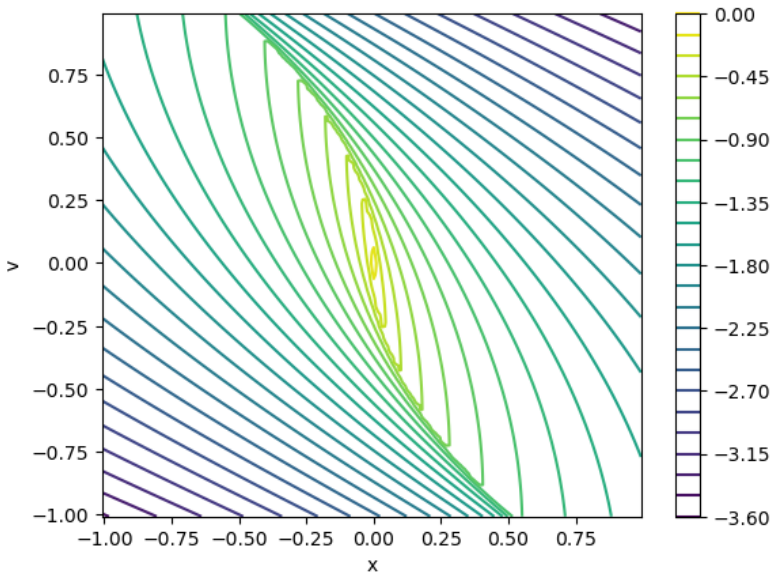


Figure 1: Contour plot of the exact value function u for the rocket car problem.

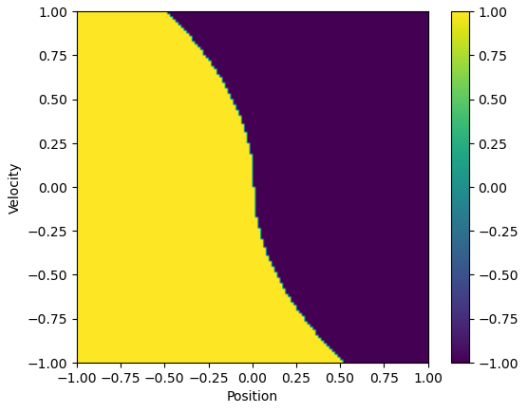


Figure 2: Optimal control for the rocket car problem.

$[(n-1),(n-1),\ldots,(n-1)]$ and a reward of 0 at all other states. Hence, the goal of the problem is to navigate towards this corner. The available actions are to increment a coordinate in the current state by ± 1 or to stay in the current state. If we reach a boundary and select an action that would take us out of the grid, we simply stay in the current state.

It is clear from the problem specification that the optimal policy is simply to increment each coordinate of the state by $+1$ until we reach the target state. The number of times we must increment corresponds to the ℓ^1 distance between the current state and the target state. Furthermore, from the Bellman optimality equation (10), we see that the optimal value function is inversely proportional to the ℓ^1 distance. We visualize the gridworld problem in Figure 3.

5 Numerical Methods

In this section, we explore numerical methods to aid us in solving two of the optimal control problems posed in the previous section. We discuss the method we use to solve the eikonal equation (the fast sweeping method) and value iteration for tabular reinforcement learning.

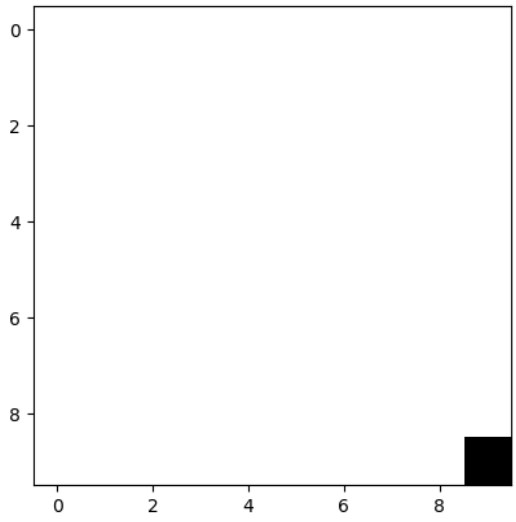


Figure 3: Illustration of the gridworld problem for $n = 10$ and $d = 2$. The bottom-right corner is shaded to indicate that it incurs a reward of 1.

5.1 The Fast Sweeping Method for the Eikonal Equation

The fast sweeping method [14] is framed by its author as an improvement over the fast marching method [10] for numerically solving the eikonal equation. Both methods discretize d-dimensional space into a grid with N^d points and update the solution outward from the target set in iterative fashion. However, due to the heapsort operation used by the fast marching method, the fast sweeping method is more efficient. The latter has an $O(N)$ time complexity, while the former is $O(N \log N)$.

We recall the setting of the eikonal equation from (11), which arose as part of finding the distance to a target set Ω. Following [14], we present the fast sweeping algorithm for the two-dimensional case, though it is applicable in any dimension. We begin with an initialization that captures the terminal condition. To do this, we initialize the grid points close to the boundary Γ to the exact solution (or an approximate interpolated solution if this is more practical). All other grid points are initialized to a high value (much higher than their actual distance to Ω). We note that [14] describes the initialization process rather loosely, suggesting that the exact criteria for "closeness" to the boundary and the quality of an interpolated solution at these grid points are flexible[2].

Let i represent a horizontal index and j represent a vertical index. For a grid with spacing h, we denote the point on the grid at index (i, j) by (x_i, y_j), the speed at this point by $F_{i,j}$, the numerical solution at this point by $u^h_{i,j}$. The fast sweeping algorithm proceeds by iterating ("sweeping") over the grid along four different orderings repeatedly until convergence:

(1) $i = 1 : N, j = 1 : N$

(2) $i = N : 1, j = 1 : N$

(3) $i = N : 1, j = N : 1$

(4) $i = 1 : N, j = N : 1$.

At each index (i, j) in a sweep, we find a "candidate" update to $u^h_{i,j}$ by solving the equation

$$\left[(\bar{u}^h_{i,j} - u^h_{xmin})^+\right]^2 + \left[(\bar{u}^h_{i,j} - u^h_{ymin})^+\right]^2 = F^2_{i,j} h^2 \tag{18}$$

for $\bar{u}^h_{i,j}$, where $u^h_{xmin} = \min(u^h_{i-1,j}, u^h_{i+1,j})$, $u^h_{ymin} = \min(u^h_{i,j-1}, u^h_{i,j+1})$, and $(\cdot)^+$ is the function returning the positive part of a real number. On the boundary of the grid, we modify the above equation

[2]Further evidence of this "looseness" can be observed in the convergence result for the fast sweeping method.

to a one-sided difference. In the two-dimensional case, [14] shows that (18) has the closed-form solution

$$\bar{u}^h_{i,j} = \begin{cases} \min(a,b) + F_{i,j}h, & |a-b| \geq F_{i,j}h \\ \frac{a+b+\sqrt{2F^2_{i,j}h^2-(a-b)^2}}{2}, & |a-b| < F_{i,j}h, \end{cases} \tag{19}$$

where $a = u^h_{xmin}$, $b = u^h_{ymin}$.

In n dimensions, [14] summarizes a procedure to recover the exact solution to an equation of the form

$$\left[(\bar{u} - a_1)^+\right]^2 + \cdots + \left[(\bar{u} - a_n)^+\right]^2 = F^2_{i,j}h^2. \tag{20}$$

Assume without loss of generality that $a_1 \leq a_2 \leq \cdots \leq a_n$, and let $a_{n+1} = \infty$. We observe that since $h^2 > 0$, there must exist $1 \leq p \leq n$ such that

$$(\bar{u} - a_1)^2 + \cdots + (\bar{u} - a_p)^2 = F^2_{i,j}h^2. \tag{21}$$

and $a_p \leq \bar{u} \leq a_{p+1}$. Once p is determined, then it suffices to solve the above quadratic equation for $\bar{u}$ and take the larger of the two solutions.

It remains to determine p, but the procedure is immediate from our discussion above. For each $1 \leq \tilde{p} \leq n$, solve the quadratic equation (21) (replace p with $\tilde{p}$) for $\bar{u}$ and take the larger of the two solutions. If the solution is in $[a_{\tilde{p}}, a_{\tilde{p}+1}]$, stop and take this solution to be $\bar{u}^h_{i_1,\ldots,i_n}$.

After solving the equation simply take $u^h_{i,j} \leftarrow \min(\bar{u}^h_{i,j}, u^h_{i,j})$.

[14] proves the following theoretical guarantee for the performance of the fast sweeping method.

Theorem 5. *If the distance function in the neighbourhood of an arbitrary data set Γ in $\mathbb{R}^n$ is given initially, let $u^h(x,\Gamma)$ be the numerical solution by the fast sweeping method after 2^n sweeps. We have*

$$\bar{u}^h(x,\Gamma) \leq u^h(x,\Gamma) \leq d(x,\Gamma) + O\big(h \log \tfrac{1}{h}\big).$$

Suppose we are in the case of constant speed (i.e., $F = 1$). The intuition behind the fast sweeping method is similar to the one underlying Dijkstra's algorithm [4] for finding the shortest distance between nodes in a graph. In the latter case, we find the distance between a source node s and some vertex v by considering the decomposition $d(s,v) \leq d(s,u) + d(u,v)$ for each neighbour u of v. Assuming the distances $d(s,u)$ are correct, then the distance $d(s,v)$ can be found by taking the minimum of $d(s,u) + d(u,v)$ over all neighbours u of v. Let u^* denote the neighbour corresponding to this minimum. Then, we can view the distance information as propagating outward from s to v via u^*.

As is indicated in [14], the above intuition carries over perfectly to the one-dimensional case of the fast sweeping method. In this case, information propagates from the boundary Γ to all points in $\mathbb{R}$. Hence, if the value u^h_i is not already correct, it will be updated via its neighbouring grid points as $\min(u^h_{i-1}, u^h_{i+1}) + h$. By sweeping over the grid from left to right and then from right to left, we ensure that we capture both possible directions of information propagation. Hence, u^h_i will be updated to the correct value at some point during the two sweeps.

In higher dimensions, there are infinitely many possible directions of information propagation (as opposed to just two in the one-dimensional case). However, as we can see in the two-dimensional case with the four sweeps that we specified, we can effectively capture this using the right sweeps and a sufficiently fine grid. For example, if Γ is the unit circle, information propagates along the directions $[\pm 1, \pm 1]^T$, where the sign is determined by the quadrant and whether we are inside or outside of the circle (see Figure 2.2 from [14]). Each of these four possible directions of propagation is linked to one of the four sweeps that we conduct in the two-dimensional case.

5.2 Value Iteration for Gridworld

Recall the Bellman optimality equation (10). This result is significant in that it gives us a recursive formulation for the optimal state-value function. We can use this to set up a fixed point iteration

$$v^{k+1}(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v^k(s')] \tag{22}$$

for all $s \in S$. This approach is referred to as *value iteration* [12].

To legitimize this, we must appeal to the Banach Fixed Point Theorem. We follow the discussion in [9].

Since we are assuming a finite state space, we encode our fixed point iteration using matrices and vectors. Let $S = \{s_1, \dots, s_n\}$. Let $V = [v(s_1), \dots, v(s_n)]^T$ denote our estimate of the state-value function at each state. For each action $a \in A$, define a matrix P^a and a vector R^a such that

$$P^a_{ij} = \mathbb{P}(S_{t+1} = s_j | S_t = s_i, A_t = a) \qquad R^a_i = \mathbb{E}[R_{t+1} | S_t = s_i, A_t = a].$$

Then, we can encode our value iteration update as

$$V_{k+1} = \max_a R^a + \gamma P^a V_k, \tag{23}$$

where the maximum is understood as being taken row-wise.

Then, we define our update operator

$$T^*(V) = \max_a R^a + \gamma P^a V. \tag{24}$$

Lemma 6. *Suppose $\gamma < 1$. Then, T^* is a contraction mapping in the ℓ^∞ norm.*

Proof. Let $U, V \in \mathbb{R}^n$ and consider the entry U_i, V_i for some $1 \leq i \leq n$. Assume without loss of generality that $T^*(U)_i \geq T^*(V)_i$. Moreover, let

$$a_i^* = \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma U(s')\big].$$

Then

$$\begin{aligned}
0 &\leq T^*(U)_i - T^*(V)_i \\
&\leq \sum_{s',r} p(s', r|s, a_i^*)[r + \gamma U(s')] - \sum_{s',r} p(s', r|s, a_i^*)[r + \gamma V(s')] \\
&= \gamma \sum_{s',r} p(s', r|s, a_i^*)[U(s') - V(s')] \\
&\leq \gamma \sum_{s',r} p(s', r|s, a_i^*)||U - V||_\infty \\
&= \gamma ||U - V||_\infty.
\end{aligned}$$

□

Theorem 7. *Suppose $\gamma < 1$. Then, value iteration converges to v_*.*

Proof. From the Bellman optimality equation, we have that v_* is a fixed point of T^*. Since T^* is a contraction mapping, we immediately deduce the result from the Banach Fixed Point Theorem. □

The above setting with a finite state space is often referred to as *tabular* reinforcement learning. It derives its name from the fact that the value function can be stored as a table of values. This is the setting of our gridworld toy problem. However, even if the state space is finite, it is often to large for storing the value function in a table to be reasonable. This can already be seen by scaling the dimension of gridworld, which leads to an exponential growth in the number of states. As another example, one cannot hope to store a value for every single position in chess.

When we move out of the tabular setting, the theory of reinforcement learning lies on shaky ground. To handle non-tabular problems, practitioners resort to function approximation [12]. That is, the value function is parametrized as v_θ and states $s \in S$ are represented by *features* $f(s) \in \mathbb{R}^d$. There is no rigorous definition of a feature – it is understood to be an encoding of a state that aids in solving the task. One desideratum is that nearby states (i.e., where one state is reachable from another with few actions) be assigned similar features (e.g., in the sense of Euclidean distance between features).

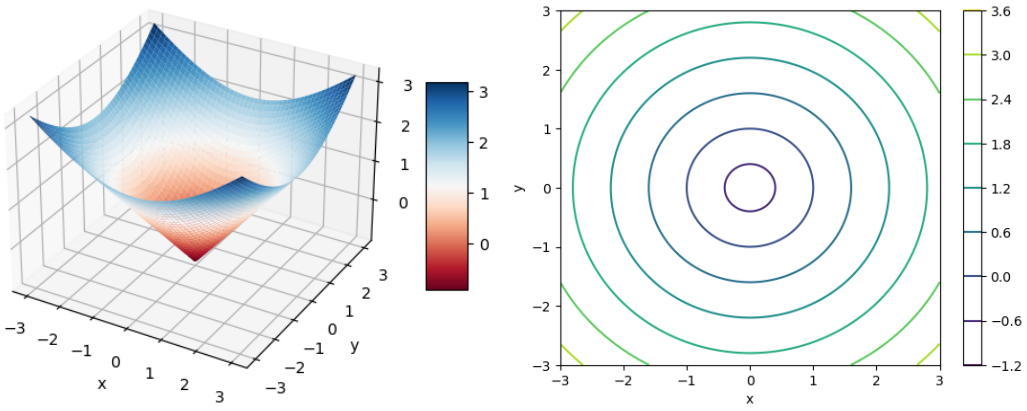


Figure 4: Numerical solution $u_h(x, y)$ to the eikonal equation for distance to the unit circle using a 1024×1024 grid with $x \in [-3, 3]$, $y \in [-3, 3]$. The graph of $u_h(x, y)$ is depicted on the left and the level curves of $u_h(x, y)$ are depicted on the right.

Convergence results do exist in the case of linear function approximators, i.e., when $v_\theta(s) = \theta^T f(s)$ for some $\theta \in \mathbb{R}^d$ [13, 12]. However, there are no such results in the nonlinear setting (i.e., when the value function is approximated using a deep neural network).

6 Experiments

In this section, we detail our experiments on the two example problems presented in Section 4 using the numerical methods presented in Section 5.

6.1 Distance to a Set

Recalling the setup from Section 4.1, we take the target set to be the closed unit ball in $\mathbb{R}^2$

$$\Omega = \{(x, y) : x^2 + y^2 \leq 1\}$$

and the boundary is thus the unit circle

$$\Gamma = \{(x, y) : x^2 + y^2 = 1\}.$$

Hence, the optimal control problem reduces to finding the (signed) distance between a point $(x, y) \in \mathbb{R}^2$ and the unit circle. Clearly, the exact solution is simply $u(x, y) = \sqrt{x^2 + y^2} - 1$.

We consider the ℓ_h^∞ error of numerical solutions (via the fast sweeping method) to the eikonal equation for this problem for different settings of the grid spacing h. The latter is determined by the number of grid points on each axis. For both the x and y-axes, we take N points from the interval $[-3, 3]$, where $N \in \{16, 32, 64, 128, 256, 512, 1024\}$. Our Python implementation of the fast sweeping method for this problem is provided in Appendix A.

Our results for the unit circle experiment are summarized in Figures 4 and 5. We empirically achieve a better convergence rate than what is predicted by the theory and can approximate the true solution to 2 digits of accuracy on the rectangle $[-3, 3] \times [-3, 3]$.

6.2 Gridworld

For our experiment, we take gridworld with $n = 10$ and $d = 2$ for a total of 100 states. We conduct value iteration by sweeping over all states repeatly and applying the aforementioned T^* operator. The progression of the algorithm, as depicted in Figure 6, gives the intuition of value propagating outward from the target set, but which gets dampened as it gets further and further away. Nearby

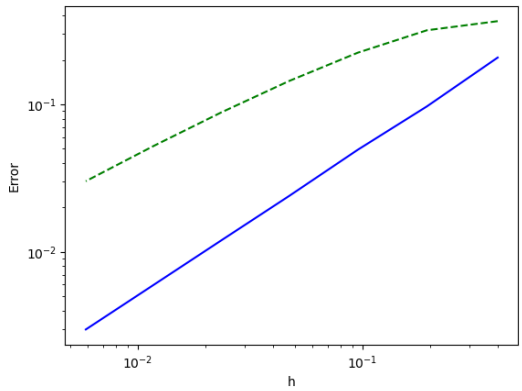


Figure 5: Convergence plot for the numerical solution $u_h(x, y)$ to the eikonal equation for distance to the unit circle. The blue line corresponds to the error of the numerical solution while the dotted green line corresponds to the asymptotic bound from Theorem 5. The error considered here is the ℓ_h^∞ error.

states to the target get nontrivial updates earlier than further states. This can be seen from the first sweep. Since we only look ahead by one step during value iteration, only the immediate neighbours of the target state will "see" the reward of 1 at the target and thus update their values. In the second step, the neighbours of the neighbours will get updated, and so on.

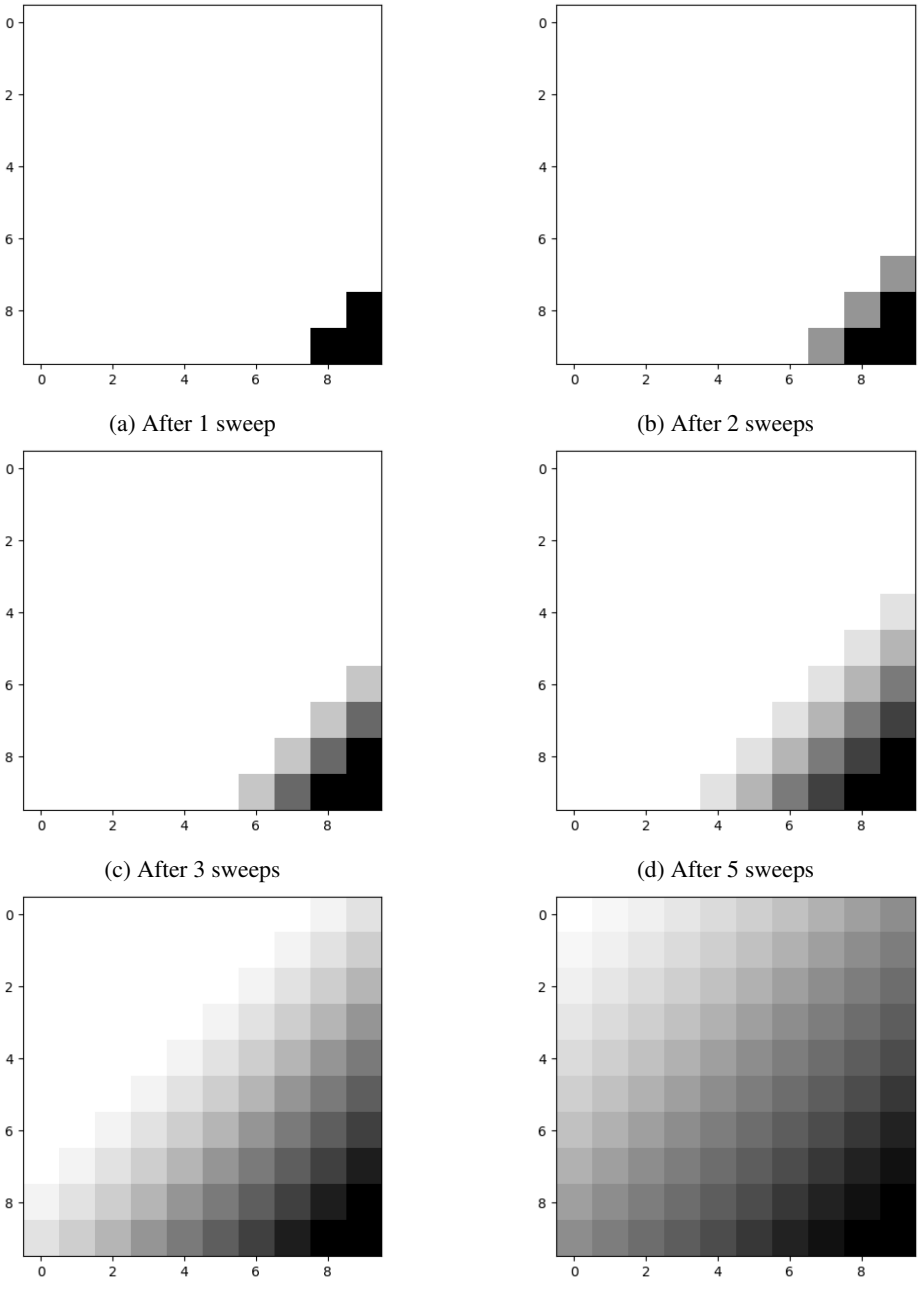
From the above perspective, we see the similarity between the fast sweeping method used for the eikonal equation and the value iteration used here. Just as in fast sweeping, the value of each grid point is adjusted based on the values of its neighbours which are nearest to the target set (i.e., which reflect the direction that distance is "propagating").

7 Conclusion

In this paper, we have explored the dynamic programming method for optimal control and reinforcement learning problems. In our particular context, we pose optimal control and reinforcement learning as being two sides of the same coin, with the former addressing the continuous-time case and the latter addressing the discrete-time case. In the optimal control setting, we discussed how the optimal control can be extracted from the value function, which in turn arises as the solution to the Hamilton-Jacobi-Bellman equation for the problem. In reinforcement learning, the optimal policy is also extracted from the value function, which is the unique solution of the Bellman equation. For the minimum time to target problem, we solve the continuous-time version with fast sweeping for the eikonal equation and the discrete-time version with value iteration. While the parallels between optimal control and reinforcement learning are instructive, we must be careful in attempting to generalize them. The difference between optimal control and reinforcement learning truly manifests when we consider reinforcement learning problems where the system dynamics are not provided, as is often the case in practice.

Acknowledgements

We wish to acknowledge the input of Prof. Jean-Christophe Nave, who has been supportive throughout the development of this project and who in particular suggested the use of fast sweeping for the eikonal equation. We also wish to thank Prof. Adam Oberman for inspiring the selection of this project topic.



(a) After 1 sweep

(b) After 2 sweeps

(c) After 3 sweeps

(d) After 5 sweeps

(e) After 10 sweeps

(f) After 20 sweeps

Figure 6: Plot of the learned value function for gridworld during value iteration after a specified number of sweeps over the state space. Darker indicates higher value.

A Implementation of the Fast Sweeping Method

We implement the fast sweeping method in Python. The following is the main "sweeping" loop which, given an ordering over indices, iterates over the grid points to update the solution value.

```
def sweep(u, h, N, i_list, j_list):
    for i in i_list:
        for j in j_list:
            if i == 0:
                a = u[i+1,j]
            elif i == N-1:
                a = u[i-1,j]
            else:
                a = min(u[i-1,j], u[i+1,j])

            if j == 0:
                b = u[i,j+1]
            elif j == N-1:
                b = u[i,j-1]
            else:
                b = min(u[i,j-1], u[i,j+1])

            if abs(a-b) >= h:
                u_bar = min(a,b) + h
            else:
                u_bar = (a+b+np.sqrt(2*(h**2)- (a-b)**2))/2

            u[i,j] = min(u[i,j], u_bar)
```

Below is the code specific to the problem of finding the distance function to the unit circle (for a variety of grid spacings).

```
N_list = [16, 32, 64, 128, 256, 512, 1024]
h_list = []
error = []

for N in N_list:
    Nx = N
    Ny = N
    x_low = -3
    x_high = 3
    dx = (x_high - x_low) / (Nx-1)
    h = dx
    h_list.append(h)
    y_high = 3
    y_low = -3
    dy = (y_high-y_low) / (Ny-1)
    x = np.arange(0,Nx) * dx + x_low
    y = np.arange(0,Ny) * dy + y_low
    xx,yy = np.meshgrid(x,y)

    u_exact = np.zeros((Nx,Ny))
    u_exact = np.sqrt(np.square(xx) + np.square(yy)) - 1
    u = np.zeros((Nx, Ny))
    u = np.where(u_exact <= 1e-2, u_exact, 100)
```

```
    i_forward = np.arange(Nx)
    j_forward = np.arange(Ny)
    i_backward = np.arange(Nx-1,-1,-1)
    j_backward = np.arange(Ny-1,-1,-1)

    for _ in range(10):
        sweep(u,h,N,i_forward,j_forward)
        sweep(u,h,N,i_backward,j_forward)
        sweep(u,h,N,i_backward,j_backward)
        sweep(u,h,N,i_forward,j_backward)

    error.append(np.max(abs(u-u_exact)))
```

B Implementation of Gridworld and Value Iteration

Below is our Python implementation of the gridworld environment. We base it on our existing implementation that we have posted in a public repository.[3]

```
import numpy as np
import matplotlib.pyplot as plt
class GridEnv():
    def __init__(self, dim=2, length=10):
        self.dim = dim
        self.length = length #Side length of grid (all sides of equal length)
        self.num_states = self.length**self.dim
        self.num_actions = 2 * self.dim + 1
        self.actions = np.arange(self.num_actions)

        self.reset()

    def reset(self):
        self.state = np.random.randint(low=0, high=self.length-1, size=(self.dim,))

    def compute_reward(self, state):
        reward = 1
        for i in range(self.dim):
            if state[i] != self.length - 1:
                reward = 0
                break

        return reward

    #Map state to a scalar value
    def hash_state(self, state):
        hash = 0
        for i in range(self.dim):
            hash += (self.length**i) * state[i]

        return int(hash)

    #Map scalar value to a state
```

[3] https://github.com/kctsiolis/rl-features/blob/master/gridworld.py

```
    def unhash_state(self, hash):
        state = np.zeros(self.dim)
        for i in range(self.dim):
            hash, state[i] = divmod(hash, self.length)

        return state

    def step(self, action):
        if action > 2 * self.dim or action < 0:
            return ValueError('Action must be in range [0, {}].'.format(self.num_actions))
        if action == self.num_actions - 1: #Do nothing
            return self.state

        coord = self.state[action // 2]
        dir = 2 * (action % 2) - 1 #1 is left, 1 is right
        if (coord == 0 and dir == -1) or (coord == self.length - 1 and dir == 1):
            return self.state
        self.state[action // 2] += dir

        return self.state

    #Look ahead without actually taking a step
    def lookahead(self, state, action):
        new_state = np.copy(state)
        if action > 2 * self.dim or action < 0:
            return ValueError('Action must be in range [0, {}].'.format(self.num_actions))
        if action == self.num_actions - 1: #Do nothing
            return new_state

        coord = state[action // 2]
        dir = 2 * (action % 2) - 1 #-1 is left, 1 is right
        if (coord == 0 and dir == -1) or (coord == self.length - 1 and dir == 1):
            return new_state
        new_state[action // 2] += dir

        return new_state
```

We implement value iteration for gridworld as follows.

```
def value_iteration(cur_state):
    updates = np.zeros(env.num_actions)
    for a in env.actions:
        next_state = env.lookahead(cur_state, a)
        updates[a] = env.compute_reward(next_state) + values[env.hash_state(next_state)]

    values[env.hash_state(cur_state)] = max(updates)

n_sweeps = 20
for _ in range(n_sweeps):
    for i in range(env.num_states):
        state = env.unhash_state(i)
        value_iteration(state)
```

References

[1] R. Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954.

[2] A. E. Bryson. Optimal control-1950 to 1985. *IEEE Control Systems Magazine*, 16(3):26–33, 1996.

[3] M. G. Crandall and P.-L. Lions. Viscosity solutions of hamilton-jacobi equations. *Transactions of the American mathematical society*, 277(1):1–42, 1983.

[4] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[5] L. C. Evans. An introduction to mathematical optimal control theory version 0.2.

[6] L. C. Evans. *Partial differential equations*, volume 19. American Mathematical Soc., 2010.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[8] A. Oberman. Nonlinear elliptic partial differential equations: Applications and numerical methods. 2016.

[9] P. Poupart. Module 6 - value iteration. *CS 886 Sequential Decision Making and Reinforcement Learning*, 2013.

[10] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.

[11] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[12] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[13] J. Tsitsiklis and B. Van Roy. Analysis of temporal-diffference learning with function approximation. *Advances in neural information processing systems*, 9, 1996.

[14] H. Zhao. A fast sweeping method for eikonal equations. *Mathematics of computation*, 74(250):603–627, 2005.