Chapter 5

Dynamic Programming

Reading

1. (Thrun) Chapter 15
2. (Sutton & Barto) Chapters 3–4
3. Optional: (Bertsekas) Chapter 1 and 4

This is the beginning of Module 2, this module is about "how to act". The first module was about "how to sense". The prototypical problem in the first module was how to assimilate the information gathered by all the sensors into some representation of the world. In the next few lectures, we will assume that this representation is good, that it is accurate in terms of its geometry (small variance of the occupancy grid) and in terms of its information (small innovation in the Kalman filter etc.). Let us also assume that it has all the necessary semantics, e.g., objects are labeled as cars, buses, pedestrians etc (we will talk about how to do this in Module 4).

The prototypical problem investigated in the next few chapters is how to move around in this world, or affect the state of this world to achieve a desired outcome, e.g., drive a car from some place A to another place B.

Our philosophy about notation Material on Dynamic Programming and Reinforcement Learning (RL), which we will cover in the following chapters, contains a lot of tiny details (much more than other areas in robotics/machine learning). These details are usually glossed over in most treatments. In the interest of simplicity, other courses or most research papers these days, develop an imprecise notation and terminology to focus on the problem. However, these details of RL matter enormously when you try to apply these techniques to real-world problems. Not knowing all the details or using imprecise terminology to think about RL is unlikely to make us good at real-world applications.

For this reason, the notation and the treatment in this chapter, and the following ones, will be a bit pedantic. We will see complicated notation and

terminology for quantities, e.g., the value function, that you might see being written very succinctly in other places. We will mostly follow the notation of Dmitri Bertsekas' book on "Reinforcement Learning and Optimal Control" (http://www.mit.edu/ dimitrib/RLbook.html). You will get used to the extra notation and it will become second nature once you become more familiar with the concepts.

5.1 Formulating the optimal control problem

Let us denote the state of a robot (and the world) by $x_k \in X \subset \mathbb{R}^n$ at the k^{th} timestep. We can change this state using a control input $u_k \in U \subset \mathbb{R}^p$ and this change is written as

$$x_{k+1} = f_k(x_k, u_k) \tag{5.1}$$

for $k = 0, 1, \ldots, T-1$ starting from some initial given state x_0. This is deterministic nonlinear dynamical system (no noise ϵ in the dynamics). We will let the dynamics f_k also be a function of time k. The time T is some time-horizon up to which we care about running the system. The state-space is X (which we will assume does not change with time k) and the control-space is U.

Recall, that we can safely assume that the system is Markov. The reason for it is as follows. If it is not, and say if x_{k+1} depends upon both x_k and the previous step x_{k-1}, then we can expand the state-space to write a new dynamics in the expanded state-space. We will follow a similar program as that of Module 1: we first describe very general algorithms (dynamic programming) for general systems (Markov Decision Processes), then specialize our methods to a restricted class of systems that are useful in practice (linear dynamical systems) and then finally discuss a very general class of systems again with more sophisticated algorithms (motion-planning).

> The central question in this chapter is how to pick a control u_k. We want to pick controls that lead to desirable trajectories of the system, e.g., results in a parallel-parked car at time T and does not collide against any other object for all times $k \in \{1, 2, \ldots, T\}$. We may also want to minimize some chosen quantity, e.g., when you walk to School, you find a trajectory that avoids a certain street with a steep uphill.

> **Finite, discrete state and control-space** In this chapter we will only consider problems with finitely-many states and controls, we will assume that the state-space X and the control-space U are finite, discrete sets.

Run-time cost and terminal cost We will take a very general view of the above problem and formalize it as follows. Consider a cost function

$$q_k(x_k, u_k) \in \mathbb{R}$$

which gives a scalar real-valued output for every pair (x_k, u_k). This models the fact that you do not want to walk more than you need to get to School, i.e., we would like to minimize q_k. You also want to make sure the trajectory actually reaches the lecture venue, we write this down as another cost $q_f(x_T)$. We want to pick control inputs $(u_0, u_1, \ldots, u_{T-1})$ such that

$$J(x_0; u_0, u_1, \ldots, u_{T-1}) = q_f(x_T) + \sum_{k=0}^{T-1} q_k(x_k, u_k) \tag{5.2}$$

is minimized. The cost $q_f(x_T)$ is called the terminal cost, it is high if x_T is not the lecture room and small otherwise. The cost q_k is called the run-time cost, it is high for instance if you have to use large control inputs, e.g., x_k is a climb.

The optimal control problem Given a system $x_{k+1} = f_k(x_k, u_k)$, we want to find control *sequences* that minimize the total cost J above, i.e., we want to solve

$$J^*(x_0) = \min_{u_k \in U, k=0,\ldots,T-1} J(x_0; u_0, \ldots, u_{T-1}) \tag{5.3}$$

It is important to realize that the function $J(x_0; u_0, \ldots, u_{T-1})$ depends upon an entire sequence of control inputs and we need to find them all to find the optimal cost $J^*(x_0)$ of, say reaching the School from your home x_0.

5.2 Dijkstra's algorithm

If the state-space X and control-space U are discrete and finite sets, we can solve (5.3) as a shortest path problem using very fast algorithms. Consider the following picture. This is what would be called a transition graph for a deterministic finite-state dynamics.

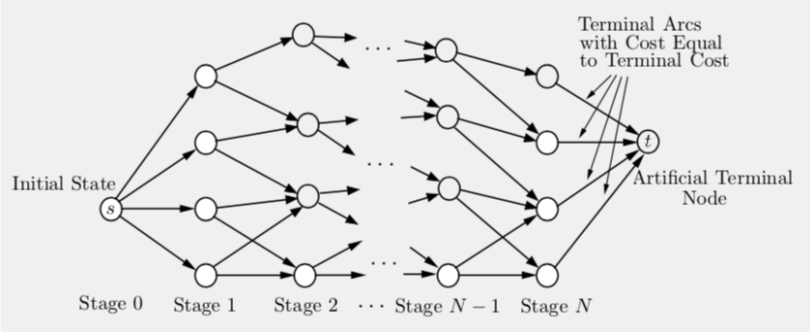


Figure 5.1: Transition graph for Dijkstra's algorithm

The graph has one source node x_0. Each node in the graph is x_k, each edge

depicts taking a certain control u_k. Depending on which control we picks, we move to some other node x_{k+1} given by the dynamics $f(x_k, u_k)$. Note that this is *not* a transition like that of a Markov chain, everything is deterministic in this graph. On each edge we write down the cost

$$\text{cost}(x_k, x_{k+1}) := q_k(x_k, u_k)$$

where $x_{k+1} = f_k(x_k, u_k)$ and "close" the graph with a dummy terminal node with the cost $q_f(x_T)$ on every edge leading to an artificial terminal node (sink).

Minimizing the cost in (5.3) is now the same as finding the shortest path in this graph from the source to the sink. The algorithm to do so is quite simple and is called Dijkstra's algorithm after Edsgar Dijkstra who used it around 1956 as a test program for a new computer named ARMAC (http://www-set.win.tue.nl/UnsungHeroes/machines/armac.html).

1. Let Q be the set of nodes that are currently unvisited; all nodes in the graph are added to it at the beginning. S is an empty set. An array called dist maintains the distance of every node in the graph from the source node x_0. Initialize $\text{dist}(x_0) = 0$ and $\text{dist} = \infty$ for all other nodes.

2. At each step, if Q is not empty, pop a node $v \in Q$ such that $v \notin S$ with the smallest $\text{dist}(v)$. Add v to S. Update the dist of all nodes u connected to v. For each u, if

$$\text{dist}(u) > \text{dist}(v) + \text{cost}(u, v)$$

update the distance of u to be $\text{dist}(v) + \text{cost}(u, v)$. If the above condition is not true do nothing.

The algorithm terminates when the set Q is empty.

You might know that there are many other variants of Dijkstra's algorithm, e.g., the A^* algorithm that are quicker to find shortest paths. We will look at some of these in the next chapter.

❓ Shortest path algorithms do not work if there are cycles in the graph because the shortest path is not unique. Are there cycles in the above graph?

❓ What should one do if the state/control space is not finite? Can we still use Dijkstra's algorithm?

> The quantity dist is quite special: observe that after Dijkstra's algorithm finishes running and the set Q is empty, the dist function gives the optimal cost to go from each node in the graph to the sink node. We wanted to only find the cost to go from source x_0 to the sink node but ended up computing the cost from every node in the graph to the sink.

5.2.1 Dijkstra's algorithm in the backwards direction

We can run Dijkstra's algorithm in the backwards direction to get the same answer as well. The sets Q and S are initialized as before. In this case we will let $\text{dist}(v)$ denote the distance of a node v to the sink node. The algorithm proceeds in the same fashion, it pops a node $v \in Q, v \notin S$ and updates the dist of all nodes u connected to v. For each u, if

$$\text{dist}(u) > \text{dist}(v) + \text{cost}(u, v)$$

then we update dist(u) to be the right-hand side of this inequality. Running Dijkstra's algorithm in reverse (from sink to the source) is completely equivalent to running it in the forward direction (from source to the sink).

ℹ If Dijkstra's algorithm (forwards or backwards) is run on a graph with n vertices and m edges, its computational complexity is $\mathcal{O}(m + n \log n)$ if we use a priority queue to find the node $v \in Q, v \notin S$ with the smallest dist. The number of edges in the transition graph in Figure 5.1 is $m = \mathcal{O}(T|X|)$.

5.3 Principle of Dynamic Programming

The principle of dynamic programming is a formalization of the idea behind Dijkstra's algorithm. It was discovered by Richard Bellman in the 1940s. The idea behind dynamic programming is quite intuitive: it says that the remainder of an optimal trajectory is optimal.

We can prove this as follows. Suppose that we find the optimal control sequence $(u_0^*, u_1^*, \ldots, u_{T-1}^*)$ for the problem in (5.3). Our system is deterministic, so this control sequence results in a *unique* sequence of states $(x_0, x_1^*, \ldots, x_T^*)$. Each successive state is given by $x_{k+1}^* = f_k(x_k^*, u_k^*)$ with $x_0^* = x_0$. The principle of optimality, or the principle of dynamic programming, states that if one starts from a state x_k^* at time k and wishes to minimize the "cost-to-go"

$$q_f(x_T) + q_k(x_k^*, u_k) + \sum_{i=k+1}^{T-1} q_i(x_i, u_i)$$

over the (now assumed unknown) sequence of controls $(u_k, u_{k+1}, \ldots, u_{T-1})$, then the optimal control sequence for this truncated problem is exactly $(u_k^*, \ldots, u_{T-1}^*)$.

The proof of the above assertion is an easy case of proof by contradiction: if the truncated sequence were not optimal starting from x_k^* there exists some other optimal sequence of controls for the truncated problem, say $(v_k^*, \ldots, v_{T-1}^*)$. If so, the solution of the original problem where one takes controls v_k^* from this new sequence for time-steps $k, k+1, \ldots, T-1$ would have a lower cost. Hence the original sequence of controls would not have been optimal.

Principle of dynamic programming. The essence of dynamic program-

ming is to solve the larger, original problem by sequentially solving the truncated sub-problems. At each iteration, Dijkstra's algorithm constructs the functions

$$J_T^*(x_T), J_{T-1}^*(x_{T-1}), \ldots, J_0^*(x_0)$$

starting from J_T^* and proceeding backwards to $J_{T-1}^*, J_{T-2}^* \ldots$. The function $J_{T-k}^*(v)$ is just the array dist(v) at iteration k of the *backwards* implementation of Dijkstra's algorithm. Mathematically, dynamic programming looks as follows.

1. Initialize $J_T^*(x) = q_f(x)$ for all $x \in X$.
2. For iteration $k = T-1, \ldots, 0$, set

$$J_k^*(x) = \min_{u_k \in U} \left\{ q_k(x, u_k) + J_{k+1}^*(f_k(x, u_k)) \right\} \tag{5.4}$$

for all $x \in X$.

After running the above algorithm we have the optimal cost-to-go $J_0^*(x)$ for each state $x \in X$, in particular, we have the cost-to-go for the initial state $J_0^*(x_0)$. If we remember the minimizer u_k^* in (5.4) while running the algorithm, we also have the optimal sequence $(u_0^*, u_1^*, \ldots, u_{T-1}^*)$. The function $J_0^*(x)$ (often shortened to simply $J^*(x)$) is the optimal cost-to-go from the state $x \in X$.

Again, we really only wanted to calculate $J_0^*(x_0)$ but had to do all this extra work of computing J_k^* for all the states.

Curse of dimensionality What is the complexity of running dynamic programming? The cost of the minimization over U is $\mathcal{O}(|U|)$, it is a bunch of comparisons between floats. The number of operations at each iteration for setting the values $J_k^*(x)$ for all $x \in X$ is $|X|$. So the total complexity is $\mathcal{O}(T\,|X|\,|U|)$.

The terms $|X|$ and $|U|$ are often the hurdle in implementing dynamic programming or any variant of it. Think of the grid-world in Problem 1 in HW 1, it had 200×200 cells which amounts to $|X| = 40,000$. This may seem a reasonable number but it explodes quickly as the dimensionality of the state-space grows. For a robot manipulator with six degrees-of-freedom, if we discretize each joint angle into 5 degree cells, the number of states is $|X| \approx$ 140 billion. The number of states $|X|$ is exponential in the dimensionality of the state-space and dynamic programming quickly becomes prohibitive beyond 4 dimensions or so. Bellman called this the *curse of dimensionality*.

Cost of dynamic programming is linear in the time-horizon Notice a very important difference between (5.4)

$$J_k^*(x) = \min_{u_k \in U} \left\{ q_k(x, u_k) + J_{k+1}^*(f_k(x, u_k)) \right\}$$

for iterations $i = T-1, \ldots, 0$ and (5.3)

$$J^*(x_0) = \min_{u_k \in U, k=0,\ldots,T-1} J(x_0; u_0, \ldots, u_{T-1}).$$

The latter has a minimization over a sequence of controls $(u_0, u_1, \ldots, u_{T-1})$ while the former has a minimization over only the control at time k, u_k over T iterations. The former is much much easier to solve because it is a sequence of $\mathcal{O}(T)$ smaller optimization problems: it is really easy to compute $\min_{u_k \in U}$) for each state x separately than to solve the gigantic minimization problem in (5.3) because in the latter case, the variable of optimization is the entire control trajectory and has size $|U|^T$.

❓ The principle of dynamic programming gives us a way to solve an optimization problem (5.3) over a really large space (the space of all control trajectories) using a linear in time-horizon number of optimization problems (5.4). Can we split any optimization problem in sub-problems like this?

Dynamic programming and Viterbi's algorithm We have seen the principle of dynamic programming in action before in Viterbi's algorithm in Chapter 2. The transition graph in Figure 5.1 is the same as the Trellis graph for Viterbi's algorithm, the run-time cost was

$$q_k(x_k, u_k) := -\log \mathrm{P}(Y_k \mid X_k) - \log \mathrm{P}(X_{k+1} \mid X_k)$$

and instead of a terminal cost q_f, we had an initial cost $-\log \mathrm{P}(X_1)$. Viterbi's algorithm computed the most likely path given observations of the HMM, i.e., the path $(X_1, \ldots, X_T)$ that maximizes the probability $\mathrm{P}(X_1, \ldots, X_T \mid Y_1, \ldots, Y_T)$ is simply the solution of dynamic programming for the Trellis graph.

❓ How should one modify dynamic programming if we have a non-additive cost, e.g., the runtime cost at time k given by q_k is a function of both x_k and x_{k-1}?

5.3.1 Q-factor

The quantity

$$Q_k^*(x, u) := q_k(x, u) + J_k^*(f_k(x, u))$$

is called the Q-factor. It is simply the expression that is minimized in the right-hand side of (5.4) and denotes the cost-to-go if control u was picked at state x (corresponding to cost $q_k(x, u)$) and the the optimal control trajectory was followed after that (corresponding to cost $J_k^*(f_k(x, u))$ from state $x' = f_k(x, u)$). This nomenclature was introduced by Watkins in his thesis.

Q-factors and the cost-to-go are equivalent ways of thinking about dynamic programming. Given the Q-factor, we can obtain the cost-to-go J_k^* as

$$J_k^*(x) = \min_{u_k \in U} Q_k^*(x, u_k). \tag{5.5}$$

which is precisely the dynamic programming update (by definition) in (5.4). We can also write dynamic programming completely in terms of Q-factors as follows.

Dynamic programming written in terms of the Q-factor

1. Initialize $Q_T^*(x,u) = q_f(x)$ for all $x \in X$ and all $u \in U$.
2. For iteration $k = T-1, \ldots, 0$, set

$$Q_k^*(x,u) = q_k(x,u) + \min_{u' \in U} Q_{k+1}^*(f_k(x,u), u'). \tag{5.6}$$

for all $x \in X$ and all $u \in U$.

As yet, it may be seem unnecessary to think of the Q-factor (which is a larger array with $|X| \times |U|$ entries) instead of the cost-to-go (which only has $|X|$ entries in the array).

Value function The following terminology is commonly used in the literature

$$\text{value function} \equiv \text{cost-to-go } J^*(x)$$
$$\text{action-value function} \equiv \text{Q-factor } Q^*(x,u).$$

Since the two functions are equivalent, we will call both as "value functions". The difference will be clear from context.

5.4 Stochastic dynamic programming: Value Iteration

Let us now see how dynamic programming looks for a Markov Decision Process (MDP). As we saw in Chapter 3, we can think of MDPs as stochastic dynamical systems denoted by

$$x_{k+1} = f_k(x_k, u_k) + \epsilon_k; \; x_0 \text{ is given.}$$

We will assume that we know the statistics of the noise ϵ_k at each time-step (say it is a Gaussian). Stochastic dynamical systems are very different from deterministic dynamical systems, given the same sequence of controls $(u_0, \ldots, u_{T-1})$, we may get different state trajectories $(x_0, x_1, \ldots, x_T)$ depending upon the realization of noise $(\epsilon_0, \ldots, \epsilon_{T-1})$. How should we find a good control trajectory then? One idea is to modify (5.3) to minimize the expected value of the cost over all possible state-trajectories

$$J(x_0; u_0, \ldots, u_{T-1}) = \mathop{\mathrm{E}}_{(\epsilon_0, \ldots, \epsilon_{T-1})} \left[q_f(x_T) + \sum_{k=0}^{T-1} q_k(x_k, u_k) \right] \tag{5.7}$$

Suppose we minimized the above expectation and obtained the value function $J^*(x_0)$ and the optimal control trajectory $(u_0^*, \ldots, u_{T-1}^*)$. As the robot starts executing this trajectory, the realized versions of the noise ϵ_k might differ a lot from their expected value, and the robot may find itself in very different states x_k than the average-case states considered in (5.10).

Draw the picture of a one-dimensional stochastic dynamical system (random walk on a line) and see that the realized trajectory of the system can be very different from the average trajectory.

Feedback controls The concept of feedback control is a powerful way to resolve this issue. Instead of seeking $u_k^* \in U$ as the solutions of (5.10), we instead seek a *function*

$$u_k(x) : X \mapsto U \tag{5.8}$$

that maps the state-space X to a control U. Effectively, given a feedback control $u_k(x)$ the robot knows what control to apply at its current realized state $x_k \in X$, namely $u_k(x_k)$, even if the realized state x_k is very different from the average-case state. Feedback controls are everywhere and are critical to using controls in the real world. For instance, when you tune the shower faucet to give you a comfortable water temperature, you are constantly estimating the state (feedback using the temperature) and turning the faucet accordingly. Doing this without feedback would leave you terribly cold or scalded. We will denote the space of all feedback controls $u_k(\cdot)$ that depend on the state $x \in X$ by

$$u_k(\cdot) \in \mathcal{U}(X).$$

Control policy A sequence of feedback controls

$$\pi = (u_0(\cdot), u_1(\cdot), \dots, u_{T-1}(\cdot)). \tag{5.9}$$

is called a control policy. This is an object that we will talk about often. It is important to remember that a control policy is set of controllers (usually feedback controls) that are executed at each time-step of a dynamic programming problem.

The **stochastic optimal control problem** finds a sequence of feedback controls $(u_0(\cdot), u_1(\cdot), \dots, u_T(\cdot))$ that minimizes

$$J(x_0; u_0(\cdot), \dots, u_{T-1}(\cdot)) = \mathop{\mathrm{E}}_{(\epsilon_0,\dots,\epsilon_{T-1})}\left[q_f(x_T) + \sum_{k=0}^{T-1} q_k(x_k, u_k(x_k))\right].$$

The value function is given by

$$J^*(x_0) = \min_{u_k(\cdot)\in\mathcal{U}(X),\ k=0,\dots,T-1} J(x_0; u_0(\cdot), \dots, u_{T-1}(\cdot)) \tag{5.10}$$

The optimal sequence of feedback controls (in short, the optimal control trajectory) is the one that achieves this minimum.

Dijkstra's algorithm no longer works, as is, if the edges in the graph are stochastic but we can use the principal of dynamic programming to write the solution for the stochastic optimal control problem. The idea remains the same, we compute a sequence of cost-to-go functions $J_T^*(x), J_{T-1}^*(x), \dots, J_0^*(x)$, and in particular $J_0^*(x_0)$, proceeding *backwards*.

Finite-horizon dynamic programming for stochastic systems.

ℹ All this sounds very tricky and abstract but you will quickly get used to the idea of feedback control because it is quite natural. You can think of feedback control as being analogous to the innovation term in the Kalman filter $K(y_k - C\mu_{k+1|k})$ which corrects the estimate $\mu_{k+1|k}$ to get a new estimate $\mu_{k+1|k+1}$ using the *current* observation y_k. Filtering would not work at all if the innovation term did not depend upon the actual observation y_k and only depended upon some average observation.

1. Initialize $J_T^*(x) = q_f(x)$ for all $x \in X$.
2. For all times $k = T-1, \ldots, 0$, set

$$J_k^*(x) = \min_{u_k(\cdot) \in \mathcal{U}(X)} \left\{ q_k(x, u_k(x)) + \mathop{\mathrm{E}}_{\epsilon_k} \left[J_{k+1}^*(f_k(x, u_k(x)) + \epsilon_k) \right] \right\} \tag{5.11}$$

for all $x \in X$.

Just like (5.4), we solve a sub-problem for one time-instant at each iteration. But observe a few importance differences in (5.11) compared to (5.4).

1. There is an expectation over the noise ϵ_k in the second term in the curly brackets. The second term in the curly brackets is the average of the cost-to-go of the truncated sub-problems from time $k+1, \ldots, T$ over all possible starting states $x' = f_k(x_k, u_k(x_k)) + \epsilon_k$. This makes sense, after taking the control $u_k(x_k)$, we may find the robot at any of the possible states $x' \in X$ depending upon different realizations of noise ϵ_k and the cost-to-go from x_k is therefore the average of the cost-to-go from each of those states (according to the principal of dynamic programming).
2. The minimization in (5.11) is performed over a function

$$\mathcal{U}(X) \ni u_k(\cdot) : X \mapsto U.$$

Since our set of states and controls is finite, this involves finding a table of size $|X| \times |U|$ for each iteration. In (5.4), we only had to search over a set of values $u_k \in U$ of size $|U|$. At the end of dynamic programming, we have a sequence of feedback controls

$$(u_0^*(\cdot), u_1^*(\cdot), \ldots, u_{T-1}^*(\cdot)).$$

Each feedback control $u_k^*(x)$ tells us what control the robot should pick if it finds itself at a state x at time k.
3. If we know the dynamical system, not in its functional form $x_{k+1} = f_k(x_k, u_k) + \epsilon_k$ but rather as a transition matrix $\mathrm{P}(x_{k+1} \mid x_k, u_k)$ (like we had in Chapter 2) then the expression in (5.11) simply becomes

$$J_k^*(x) = \min_{u_k(\cdot) \in \mathcal{U}(X)} \left\{ q_k(x, u_k(x)) + \mathop{\mathrm{E}}_{x' \sim \mathrm{P}(\cdot \mid x_k, u_k(x_k))} \left[J_{k+1}^*(x') \right] \right\} \tag{5.12}$$

❓ Why should we only care about minimizing the average cost in the objective in (5.10)? Can you think of any other objective we may wish to use?

Computational complexity The form in (5.12) helps us understand the computational complexity, each sub-problem performs $|X| \times |X| \times |U|$ amount of work and therefore the total complexity of stochastic dynamic programming is

$$\mathcal{O}(T|X|^2|U|).$$

Naturally, the quadratic dependence on the size of the state-space is an even bigger hurdle while implementing dynamic programming for stochastic systems.

5.4.1 Infinite-horizon problems

In the previous section, we put a lot of importance on the horizon T for dynamic programming. This is natural: if the horizon T changes, say you are in a hurry to get to school, the optimal trajectory may take control inputs that incur a lot of runtime cost simply to reach closer to the goal state (something that keeps the terminal cost small). In most, real-world problems, it is not very clear what value of T we should pick. We therefore formulate the dynamic programming problem as something that also allows a trajectory of infinite steps but also encourages the length of the trajectory to be small enough in order to be meaningful. Such problems are called infinite-horizon problems ($T \to \infty$).

Stationary dynamics and run-time cost We think of infinite-horizon problems in the following way: at any time-step, the length of the trajectory *remaining* for the robot to traverse is infinite. It helps in this case to solve a restricted set of problems where the system dynamics and run-time cost do not change as a function of time (they only change as a function of the state and the control). We will set

$$\begin{aligned} q(x,u) &\equiv q_k(x,u), \\ f(x,u) &\equiv f_k(x,u) \end{aligned}$$

for all $x \in X$ and $u \in U$. Such a condition is called stationarity. If the system is stochastic, we also require that the distribution of noise ϵ_k does not change as a function of time (it could change in (5.11) but we did not write it so). The infinite-horizon setting is never quite satisfied in practice but it is a reasonable formulation for problems that run for a long length of time.

Infinite-horizon objective The objective that we desire be minimized by an infinite-horizon control policy

$$\pi = (u_0(\cdot), u_1(\cdot), \dots, u_T(\cdot), u_{T+1}(\cdot), \dots,)$$

is defined in terms of an asymptotic limit

$$J(x_0;\pi) = \lim_{T\to\infty} \mathop{\mathrm{E}}_{(\epsilon_0,\dots,\epsilon_{T-1})} \left[\sum_{k=0}^{T-1} \gamma^k q(x_k, u_k(x_k)) \right]. \tag{5.13}$$

and we again wish to solve for the optimal cost-to-go

$$J^*(x_0) = \operatorname*{argmin}_{\pi} J^*(x_0;\pi). \tag{5.14}$$

Thus the infinite horizon costs of a policy is the limit of its finite horizon costs as the horizon tends to infinity. Notice a few important differences when

compared to (5.7).

1. The objective is a limit, it is effectively the cost of the trajectory as it is allowed to stretch for a larger and larger time-horizon.
2. There is no terminal cost in the objective function; this makes sense because an explicit terminal state x_T does not exist anymore. In infinite-horizon problems, you should think of the terminal cost as being incorporated inside the run-time cost $q(x, u)$ itself, e.g., move the robot to minimize the fuel used at *this* time instant but also move it in a way that it reaches the goal *at some time in the future*.
3. **Discount factor—** Depending upon what controls we pick, the summation
$$\sum_{k=0}^{T} q(x_k, u_k(x_k))$$
can diverge to infinity as $T \to \infty$ and thereby a meaningful solution to the infinite-horizon problem may not exist. In order to avoid this, we use a scalar
$$\gamma \in (0, 1)$$
known as the discount factor in the formulation. It puts more emphasis on costs incurred earlier in the trajectory than later ones and thereby encourages the length of the trajectory to be small. Notice that $\sum_{k=0}^{\infty} \alpha^k = 1/(1-\alpha)$ if $|a| < 1$, so if the cost $|q(x_k, u_k(x_k))| < 1$, then we know that the objective in (5.13) always converges.

Stochastic shortest path problems It is important to remember that the discount factor is chosen by the user, no one prescribes it. There is also a class of problems where we may choose $\gamma = 1$ but in these cases, there should exist some *essentially terminal* state in the state space where we can keep taking a control such that the runtime cost $q(x, u)$ is zero. Otherwise, the objective will diverge. The goal region in the grid-world problem could be an example of such state. Such problems are called stochastic shortest path problems because the time-horizon is not *actually* infinite, we just do not know how many time-steps it will take for the robot to go to the goal location. Naturally, stochastic shortest path problems are a generalization of the shortest path problem solved by Dijkstra's algorithm. The algorithms we discuss next will work for such problems.

Stationary policy It seems a bit cumbersome to carry around an infinitely long sequence of feedback controls in infinite-horizon problems. Since there is an infinitely-long trajectory yet to be traveled *at any given time-step*, the optimal control action that we take should only depend upon the current state. This is indeed true mathematically. If $J^*(x)$ is the optimal cost-to-go in the infinite-horizon problem starting from a state x, using the principle of dynamic programming, we should also have that we can split this cost as the best one-step cost of the current state x added to the optimal cost-to-go from the state $f(x, u)$ realized after taking the optimal control u:

$$J^*(x) = \min_{u(x) \in \mathcal{U}(X)} \mathop{\mathrm{E}}_{\epsilon} \left[q(x, u(x)) + J^*(f(x, u(x)) + \epsilon) \right]. \tag{5.15}$$

We will study this equation in depth soon. But if we find the minimum at $u^*(x)$ for this equation, then we can run the policy

$$\pi^* = (u^*(\cdot), u^*(\cdot), \ldots, u^*(\cdot), \ldots)$$

for the entire infinite horizon. Such a policy is called a stationary policy. Intuitively, since the future optimization problem (tail of dynamic programming) from a given state x looks the same regardless of the time at which we start, optimal policies for the infinite-horizon problem can be found even inside the restricted class of policies where the feedback control does not change with time k.

We will almost exclusively deal with stationary policies in this course.

5.4.2 Dynamic programming for infinite-horizon problems

We wish to compute the optimal cost-to-go of starting from a state x and taking an infinitely long trajectory that minimizes the objective (5.13). We will exploit the equation in (5.15) and develop an iterative algorithm to compute the optimal cost-to-go $J^*(x)$.

Value Iteration. The algorithm proceeds iteratively to maintain a sequence of approximations

$$\forall x \in X, \quad J^{(0)}(x), J^{(1)}(x), J^{(2)}(x), \ldots,$$

to the optimal value function $J^*(x)$. Such an algorithm is called "value iteration".

1. Initialize $J^{(0)}(x) = 0$ for all $x \in X$.
2. Update using the Bellman equation at each iteration, i.e., for $i = 1, 2, \ldots, N$, set

$$J^{(i+1)}(x) = \min_{u \in U} \mathop{\mathrm{E}}_{\epsilon} \left[q(x,u) + \gamma J^{(i)}(f(x,u) + \epsilon) \right]. \tag{5.16}$$

   for all $x \in X$ until the value function converges at all states, e.g.,

$$\forall x \in X, \quad |J^{(i)}(x) - J^{(i+1)}(x)| < \text{small tolerance}.$$

3. Compute the feedback control and the stationary policy $\pi^* = (u^*(\cdot), \ldots,)$ corresponding to the value function estimate $J^{(N)}$ as

$$u^*(x) = \operatorname*{argmin}_{u \in U} \mathop{\mathrm{E}}_{\epsilon} \left[q(x,u) + \gamma J^{(N)}(f(x,u) + \epsilon) \right] \tag{5.17}$$

   for all $x \in X$.

If the dynamics is given as a transition matrix, we can replace the expectation over noise E_ϵ as an expectation over the next state $x' \sim \mathrm{P}(x' \mid x, u(x))$ in (5.16) to run value iteration. Everything else remains the same

Let us observe a few important things in the above sequence of updates. First, at each iteration, we are updating the values of all $|X|$ states. This involves $|X|^2|U|$ amount of work per iteration. How many such iterations N

do we need until the value function converges? We will see in a bit, that

$$\forall x \in X, \quad J^*(x) = \lim_{N \to \infty} J^{(N)}(x).$$

Again, we really only wanted to compute the cost-to-go $J^*(x_0)$ from some initial state x_0 but computed the value function at all states $x \in X$.

Q-Iteration Just like we wrote dynamic programming in terms of the Q-factor, we can also write value iteration to find the optimal Q-factor $Q^*(x, u)$, i.e., the optimal cost-to-go of the infinitely-long trajectory that starts at state x, takes a control u at the first time-step and therefore follows the optimal policy.

1. We can again initialize $Q^{(0)}(x, u) = q(x, u)$ for all $x \in X$ and $u \in U$.
2. The Bellman update in terms of the Q-factor becomes

$$Q^{(i+1)}(x, u) = \mathop{\mathrm{E}}_{\epsilon}\left[q(x, u) + \gamma \min_{u' \in U} Q^{(i)}(f(x, u) + \epsilon, u')\right] \quad (5.18)$$

   and update this for all $x \in X$ and all $u \in U$.
3. The feedback control is the control at that state that minimizes the Q-factor

$$\forall x \in X, \quad u^*(x) = \mathop{\mathrm{argmin}}_{u' \in U} \; Q^{(N)}(x, u') \quad (5.19)$$

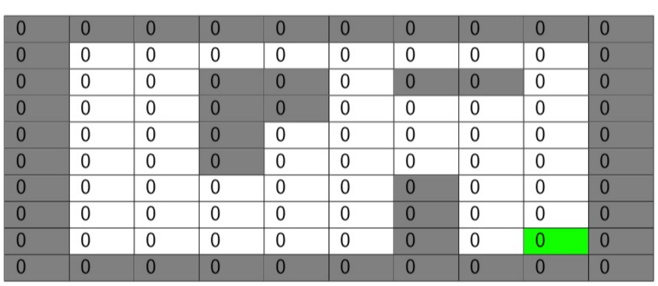
   and the control policy is

$$\pi^* = (u^*, u^*, \ldots,)$$

   Notice how we can directly find the u' that has the smallest value of $Q^{(N)}$ and set it to be our feedback control.

5.4.3 An example

Let us consider a grid-world example. A robot would like to reach a goal region (marked in green) and we are interested in computing the cost-to-go from different parts of the domain. Gray cells are obstacles that the robot cannot enter. At each step the robot can move in four directions (north, east, west, south) with a small dynamics noise which keeps it at the original cell in spite of taking the control. These pictures are when the run-time-cost is *negative*, i.e., the robot gets a certain reward $q(x, u)$ for taking the control u at cell x. Dynamic programming (and value iteration) also works in this case and we simply replace all minimizations by maximizations in the equations.

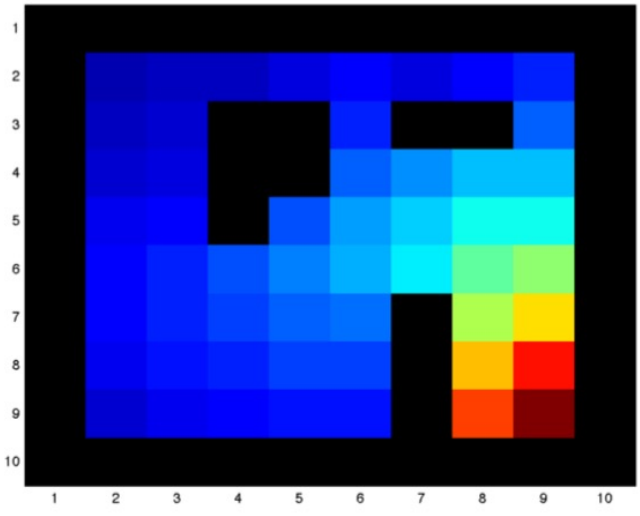


0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0.75	0
0	0	0	0	0	0	0	0.75	1	0
0	0	0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0.51	0
0	0	0	0	0	0	0	0.56	1.43	0
0	0	0	0	0	0	0	1.43	1.9	0
0	0	0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0	0	0
0	0.44	0.54	0.59	0.82	1.15	0.85	1.09	1.52	0
0	0.59	0.69	0	0	1.52	0	0	2.13	0
0	0.75	0.90	0	0	2.12	2.55	2.98	3.00	0
0	0.95	1.18	0	2.00	2.70	3.22	3.80	3.88	0
0	1.20	1.55	1.87	2.41	2.92	3.51	4.52	5.00	0
0	1.15	1.47	1.74	2.05	2.25	0	5.34	6.47	0
0	0.99	1.26	1.49	1.72	1.74	0	6.69	8.44	0
0	0.74	0.99	1.17	1.34	1.27	0	7.96	9.94	0
0	0	0	0	0	0	0	0	0	0

The final value function after 50 iterations looks as follows.



5.4.4 Some theoretical results on value iteration

We list down some very powerful theoretical results for value iteration. These results are valid under a very general set of conditions and make value iteration

work for a large number of real-world problems; they are at the the heart of all modern algorithms. We will not derive them (it is easy but cumbersome) but you should commit them to memory and try to understand them intuitively.

Value iteration converges. Given *any* initialization $J^{(0)}(x)$ for all $x \in X$, the sequence of value iteration estimates $J^{(i)}(x)$ converges to the optimal cost

$$\forall x \in X, \quad J^*(x) = \lim_{N \to \infty} J^{(N)}(x)$$

The solution is unique. The optimal cost-to-go $J^*(x)$ of (5.14) satisfies the Bellman equation

$$J^*(x) = \min_{u \in U} \mathop{\mathrm{E}}_{\epsilon} \left[q(x,u) + \gamma J^*(f(x,u) + \epsilon) \right].$$

The function J^* is also the *unique* solution of this equation. In other words, if we find some other function $J'(x)$ that satisfies the Bellman equation, we are guaranteed that J' is indeed the optimal cost-to-go.

Policy evaluation: Bellman equation for a particular policy. Consider a stationary policy $\pi = (u(\cdot), u(\cdot), \ldots)$. The cost of executing this policy starting from a state x, is $J(x;\pi)$ from (5.13), also denoted by $J^\pi(x)$ for short. It satisfies the equation

$$J^\pi(x) = q(x, u(x)) + \gamma \mathop{\mathrm{E}}_{\epsilon} \left[J^\pi(f(x, u(x)) + \epsilon) \right] \tag{5.20}$$

and is also the unique solution of this equation. In other words, if we have a policy in hand, and wish to find the cost-to-go of this policy, i.e., "evaluate the policy" we can initialize $J^{(0)}(x) = 0$ for all $x \in X$ and perform the sequence of iterative updates to this initialization

$$J^{(i+1)}(x) = q(x, u(x)) + \gamma \mathop{\mathrm{E}}_{\epsilon} \left[J^{(i)}(f(x, u(x)) + \epsilon) \right]. \tag{5.21}$$

As the number of updates goes to infinity, the iterate converges to $J^\pi(x)$

$$\forall x \in X, \quad J^\pi(x) = \lim_{N \to \infty} J^{(N)}(x).$$

Policy evaluation is equivalent to solving a linear system of equations. The min operation in (5.16) or (5.18) is particularly problematic in implementing value iteration. As compared to it, observer that the corresponding equation for policy equation (5.20) does not have min operation. This allows us to write the updates in (5.21) as the solution of a linear system of equations. Since we are in a finite state-space, we can write the cost-to-go as a large vector

$$J^\pi := \left[J^\pi(x_1), J^\pi(x_2), \ldots, J^\pi(x_n) \right]^\top$$

where n is the number of total states in the state-space. We create a similar vector for the run-time cost term

$$q^u := \left[q(x_1, u(x_1)), q(x_2, u(x_2)), \ldots, q(x_n, u(x_n)) \right].$$

We know that the expectation over noise ϵ is equivalent to an expectation over the next state of the system, let us rewrite the dynamics part $f(x, u(x)) + \epsilon$ in terms of the Markov transition matrix

$$T_{x,x'} = \mathrm{P}(x' \mid x, u(x))$$

as

$$\gamma \mathop{\mathrm{E}}_{\epsilon} [J^\pi(f(x, u(x)) + \epsilon)] = \gamma \sum_{x'} T_{x,x'} J^\pi(x') = \gamma \, T J^\pi$$

to get a linear system

$$J^\pi = q^u + \gamma T J^\pi \tag{5.22}$$

which can be solved easily for $J^\pi = (I - \gamma \, T)^{-1} \, q^u$ to get the cost-to-go of a particular control policy π.

5.5 Stochastic dynamic programming: Policy Iteration

Value iteration converges exponentially quickly, but asymptotically. The number of states $|X| = n$ is finite and so is the number of controls $|U|$. This seems very funny, one would expect that we should be able to find the optimal cost $J^*(x)$ in finite time if the problem is finite, after all we need to find $|X|$ numbers $J^*(x_1), \ldots, J^*(x_n)$. This intuition is rightly placed indeed and in this section, we will discuss an algorithm called policy iteration, which is a more efficient version of value iteration.

The idea behind policy iteration is quite simple: given a stationary policy for an infinite-horizon problem $\pi = (u(\cdot), \ldots, u(\cdot))$, we can evaluate this policy to obtain its cost-to-go $J^\pi(x)$. If we now set the feedback control to be

$$\tilde{u}(x) = \operatorname*{argmin}_{u \in U} \mathop{\mathrm{E}}_{\epsilon} [q(x, u) + \gamma J^\pi(f(x, u) + \epsilon)], \tag{5.23}$$

i.e., we construct a *new control policy* that executes this new control $\tilde{u}(\cdot)$ at the first step and thereafter executes the old feedback control $u(\cdot)$

$$\pi^{(1)} = (\tilde{u}(\cdot), u(\cdot), \ldots),$$

then the cost-to-go of policy $\pi^{(1)}$ is always better:

$$\forall x \in X, \quad J^{\pi^{(1)}}(x) \leq J^\pi(x).$$

We don't have to stop at one time-step, we can patch the old policy at the first two time-steps to get

$$\pi^{(2)} = (\tilde{u}(\cdot), \tilde{u}(\cdot), \ldots),$$

and have by the same logic

$$\forall x \in X, \quad J^{\pi^{(2)}}(x) \leq J^{\pi^{(1)}}(x) \leq J^\pi(x).$$

❓ Why? It is simply because (5.23) is at least an improvement upon the feedback control $u(\cdot)$. The cost-to-go cannot improve only if the old feedback control $u(\cdot)$ where optimal to begin with.

If we build a new stationary policy

$$\tilde{\pi} = (\tilde{u}(\cdot), \tilde{u}(\cdot), \tilde{u}(\cdot), \ldots), \tag{5.24}$$

we similarly have

$$\forall x \in X, \quad J^{\tilde{\pi}}(x) \leq J^{\pi}(x).$$

This suggests an iterative way to compute the optimal stationary policy π^* starting from some initial stationary policy (i.e., implicitly a feedback controller).

Policy Iteration The algorithm proceeds to maintain a sequence of stationary policies

$$\pi^{(k)} = (u^{(k)}(\cdot), u^{(k)}(\cdot), u^{(k)}(\cdot), \ldots)$$

that converges to the optimal policy π^*.

Initialize $u^{(0)}(x) = 0$ for all $x \in X$. This gives the initial stationary policy $\pi^{(0)}$. At each iteration $k = 1, \ldots,$ we do the following two things.

1. **Policy evaluation** Use multiple iterations of (5.21) to evaluate the old policy $\pi^{(k)}$. Initialize $J^{(0)}(x) = 0$ for all $x \in X$ and iterate upon

   $$J^{(i+1)}(x) = q(x, u^{(i)}(x)) + \gamma \mathop{\mathrm{E}}_{\epsilon} \left[J^{(i)}(f(x, u^{(i)}(x) + \epsilon)) \right]$$

   for all $x \in X$ until convergence. In practice, we always use the linear system of equations in (5.22) to solve for $J^{\pi^{(k)}}$ directly.
2. **Policy improvement** Update the feedback controller using (5.23) to be

   $$u^{(k+1)}(x) = \operatorname*{argmin}_{u \in U} \mathop{\mathrm{E}}_{\epsilon} \left[q(x, u) + \gamma J^{\pi^{(k)}}(f(x, u) + \epsilon) \right]$$

   for all $x \in X$ and compute the updated stationary policy

   $$\pi^{(k+1)} = (u^{(k+1)}(\cdot), u^{(k+1)}(\cdot), \ldots)$$

The algorithm terminates when the controller does not change *at any state*, i.e., when the following condition is satisfied

$$\forall x \in X, \quad u^{(k+1)}(x) = u^{(k)}(x).$$

Just like value iteration converges to the optimal value function, it can be shown that policy iteration produces a sequence of improved policies

$$\forall x \in X, \quad J^{\pi(k+1)}(x) \leq J^{\pi(k)}(x)$$

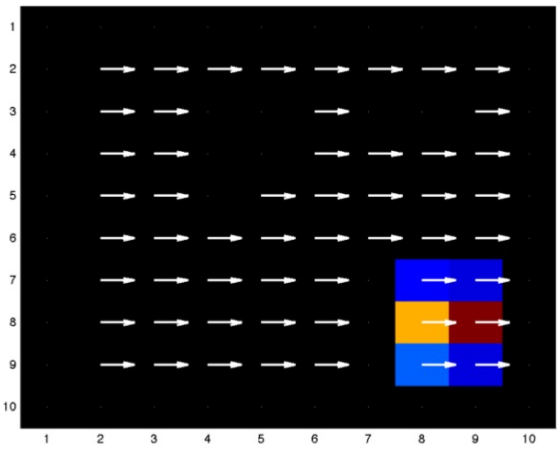
and converges to the optimal cost-to-go

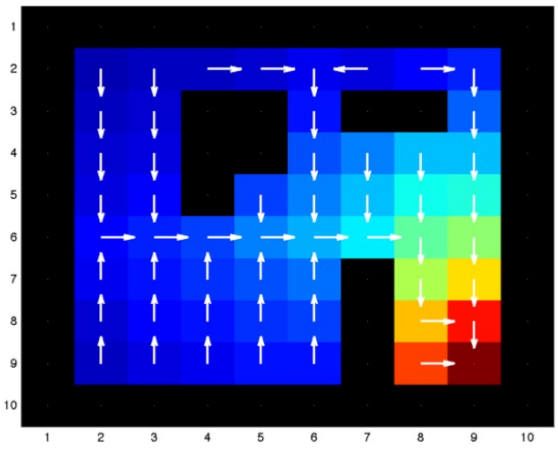
$$\forall x \in X, \quad J^*(x) = \lim_{N \to \infty} J^{\pi^{(N)}}(x).$$

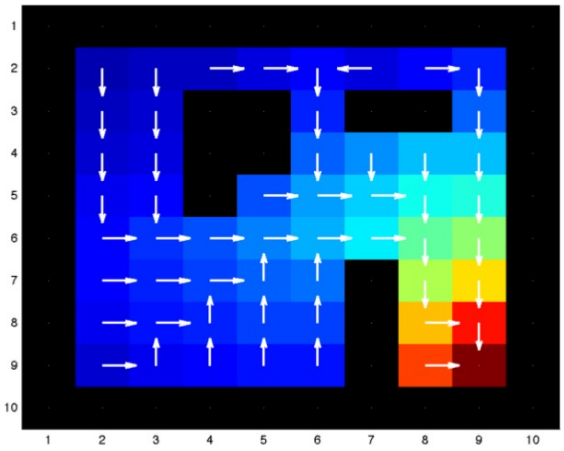
The key property of policy iteration is that we need a finite number of updates to the policy to find the optimal policy. Notice that this does not mean always mean that we are doing less work than value iteration in policy iteration. Observe that the policy evaluation step in the policy iteration algorithm performs a number of Bellman equation updates. But typically, it is observed in practice that policy iteration is much cheaper computationally than value iteration.

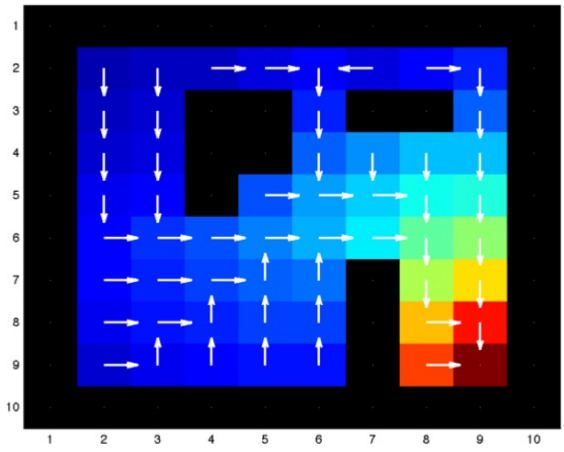
5.5.1 An example

Let us go back to our example for value iteration. In this case, we will visualize the controller $u^{(k)}(x)$ at each cell x as arrows pointing to some other cell. The cells are colored by the value function for that particular stationary policy.









The evaluated value for the policy after 4 iterations is optimal, compare this to the example for value iteration.

0	0	0	0	0	0	0	0	0	0
0	0.45	0.56	0.61	0.84	1.17	0.87	1.11	1.5411	0
0	0.61	0.71	0	0	1.54	0	0	2.16	0
0	0.78	0.93	0	0	2.16	2.59	3.02	3.03	0
0	0.98	1.21	0	2.03	2.74	3.26	3.84	3.91	0
0	1.23	1.58	1.90	2.44	2.95	3.54	4.56	5.03	0
0	1.18	1.50	1.78	2.09	2.28	0	5.38	6.51	0
0	1.02	1.29	1.52	1.76	1.77	0	6.74	8.49	0
0	0.76	1.02	1.20	1.37	1.30	0	8.01	10	0
0	0	0	0	0	0	0	0	0	0