CME 307 Course Project III: Value-Iteration Method for MDP

Johann Gaebler

March 15, 2021

Prompt. As described in Lectures, Rainforcement Learning (RL) and Markov Decision Processes (MDP) provide a mathematical framework for modeling sequential decision making in situations where outcomes are partly random and partly under the control of a decision maker. MDPs are useful for studying a wide range of optimization problems solved via Dynamic Programming (DP), where it was known at least as early as the 1950s (cf. Shapley 1953, Bellman 1957). Modern applications include dynamic planning, reinforcement learning, social networking, and almost all other dynamic/sequential decision game strategy making problems in Mathematical, Physical, Management and Social Sciences.

As described in class, the MDP problem with m states and total n actions can be formulated as a standard form linear program with m equality constraints and n variables:

$$\begin{array}{lcccccc} \min_{\mathbf{x}} & \sum_{j\in\mathcal{A}_1} c_j x_j + & \dots & +\sum_{j\in\mathcal{A}_m} c_j x_j & & \\ \text{s.t.} & \sum_{j\in\mathcal{A}_1}(\mathbf{e}_1-\gamma\mathbf{p}_j)x_j + & \dots & +\sum_{j\in\mathcal{A}_m}(\mathbf{e}_m-\gamma\mathbf{p}_j)x_j & = & \mathbf{e}, \\ & \dots & x_j & \dots & \geq & 0,\ \forall j, \end{array} \tag{1}$$

where $\mathcal{A}_i$ represents the set of all actions available in state i, $\mathbf{p}_j$ is the state transition probabilities from state i to all states and c_j is the immediate cost when action j is taken, and $0 < \gamma < 1$ is the discount factor. Also, $\mathbf{e} \in \mathbb{R}^m$ is the vector of ones, and $\mathbf{e}_i$ is the unit vector with 1 at the i-th position and zeros everywhere else. Variable x_j, $j \in \mathcal{A}_i$, is the state-action frequency or flux, or the expected present value of the number of times in which the process visits state i and takes state-action $j \in \mathcal{A}_i$. Thus, solving the problem entails choosing a state-action frequencies/fluxes that minimize the expected present value sum of total costs. The dual of the LP is

$$\begin{array}{lccl} \max_{\mathbf{y}} & \mathbf{e}^\top\mathbf{y} = \sum_{i=1}^m y_i & & \\ \text{s.t.} & y_1 - \gamma\mathbf{p}_j^\top\mathbf{y} & \leq & c_j,\ j\in\mathcal{A}_1 \\ & & \vdots & \\ & y_i - \gamma\mathbf{p}_j^\top\mathbf{y} & \leq & c_j,\ j\in\mathcal{A}_i \\ & & \vdots & \\ & y_m - \gamma\mathbf{p}_j^\top\mathbf{y} & \leq & c_j,\ j\in\mathcal{A}_m. \end{array} \tag{2}$$

where y_i represents the cost-to-go value in state i.

In this project, you may generate a Maze Game in 2D by assign actions with different costs and probability distributions to test your algorithms.

Preliminaries. We begin by establishing some notation and showing that the subsequent problems are well-posed. Let

$$\mathbf{P} = \begin{bmatrix} \mathbf{p}_1{}^\top \\ \vdots \\ \mathbf{p}_n{}^\top \end{bmatrix}$$

be the $n \times m$ matrix where row j corresponds to the transition probabilities of action j. Let

$$\mathbf{J} = \begin{bmatrix} \mathbf{e}_1^\top \\ \vdots \\ \mathbf{e}_i^\top \\ \vdots \\ \mathbf{e}_m^\top \end{bmatrix}$$

be the $n \times m$ matrix where row j equals $\mathbf{e}_i^\top$ exactly when $j \in \mathcal{A}_i$. Lastly, let $\mathbf{c}$ be the vector of immediate costs. Then, we can more compactly state our linear program as follows:[1]

$$\begin{aligned} \min \quad & \mathbf{c}^\top \mathbf{x} \\ \text{s.t.} \quad & (\mathbf{J} - \gamma \mathbf{P})^\top \mathbf{x} = \mathbf{e} \\ & \mathbf{x} \geq 0. \end{aligned}$$

Now, note that $\mathbf{J} - \gamma\mathbf{P}$ has rank m. For, trivially, for all $\mathbf{x} \in \mathbb{R}^m$, $\|\mathbf{J}\mathbf{x}\|_\infty = \|\mathbf{x}\|_\infty$. Likewise, since $\|\mathbf{p}_j\|_1 = 1$, it follow's from Hölder's inequality that

$$\|\mathbf{P}\mathbf{x}\|_\infty = \max_{j=1}^{n} |\mathbf{p}_j^\top \mathbf{x}| \leq \max_{j=1}^{n} \|\mathbf{p}_j\|_1 \cdot \|\mathbf{x}\|_\infty = \|\mathbf{x}\|_\infty.$$

Therefore, in particular, we conclude by the triangle inequality that

$$\|(\mathbf{J} - \gamma\mathbf{P})\mathbf{x}\|_\infty \geq \|\mathbf{x}\|_\infty - \gamma\|\mathbf{x}\|_\infty = (1-\gamma)\|\mathbf{x}\|_\infty.$$

Since $0 < \gamma < 1$, it follows that $\|\mathbf{x}\|_\infty \neq 0$ unless $\mathbf{x} = \mathbf{0}$, i.e., $\mathbf{J} - \gamma\mathbf{P}$ has full column rank. Thus, it follows that $(\mathbf{J} - \gamma\mathbf{P})^\top$ has rank m.

It follows, as a consequence of the Fundamental Theorem of Linear Programming [LY16, p. 21], that if there is a solution to our linear program, then there is a basic solution, and if there is an optimal solution, then there is an optimal basic solution.

Now, to see that there is a solution, let $\pi(i) \in \mathcal{A}_i$ be an arbitrary policy. Clearly the $m \times m$ matrix

$$\mathbf{W} = \begin{bmatrix} \mathbf{p}_{\pi(1)}^\top \\ \vdots \\ \mathbf{p}_{\pi(m)}^\top \end{bmatrix}$$

is stochastic, i.e., its row sums are all 1. Moreover, if $\mathbf{p}$ is any probability vector (i.e., $\|\mathbf{p}\|_1 = 1$) then trivially so is $\mathbf{W}\mathbf{p}$. Since $\mathbf{W}$ is continuous, it follows by Brouwer's Fixed Point Theorem that $\mathbf{W}$ has a fixed point, i.e., there is a (not necessarily unique) stationary distribution $\mathbf{p}^*$ such that $\mathbf{W}\mathbf{p}^* = \mathbf{p}^*$.

[1]Note that we assume here and throughout, without loss of generality, that the action sets are disjoin, i.e., $\mathcal{A}_{i_0} \cap \mathcal{A}_{i_1} = \emptyset$ for $i_0 \neq i_1$, and that if $i_0 < i_1$ and $j_0 \in \mathcal{A}_{i_0}$, $j_1 \in \mathcal{A}_{i_1}$, then $j_0 < j_1$. This can easily be arranged by copying and reordering rows of $\mathbf{P}$ and $\mathbf{J}$ and entries of $\mathbf{c}$.

Question 1. *Prove that in Eq.* (1) *every basic feasible solution represent a policy, i.e., the basic variables have exactly one variable from each state i. Furthermore, prove each basic variable value is no less than* 1, *and the sum of all basic variable values is $\frac{m}{1-\gamma}$.*

Answer. Suppose our basic optimal solution corresponds to columns $j(1) < \ldots < j(m)$ of $(\mathbf{J} - \gamma\mathbf{P})^\top$. We seek to show that $j(i) \in \mathcal{A}_i$. Our basic solution is $\sum_{i=1}^m \mathbf{x}_{j(i)} e_{j(i)}$ for $x_{j(i)} > 0$ for $i = 1, \ldots, m$. By the constraint of our linear program, it follows that

$$\sum_{i=1}^m x_{j(i)}(\mathbf{J} - \gamma P)^\top \mathbf{e}_{j(i)} = \mathbf{e},$$

i.e., for all $i = 1, \ldots, m$,

$$1 = \left[\sum_{1 \le k \le m : j(k) \in \mathcal{A}_i} x_{j(k)}\right] - \gamma \left[\sum_{k=1}^m x_{j(k)} p_{j(k),i}\right], \tag{3}$$

where $p_{j,i}$ represents the i-th entry of $\mathbf{p}_j$. Now, γ, $x_{j(k)}$, and $p_{j(k),i}$ are all positive, and so the subtrahend, $\gamma \sum_{k=1}^n x_{j(k)} p_{j(k),i}$, is positive. Consequently, if $j(k) \notin \mathcal{A}_i$ for $k = 1, \ldots, m$, then the difference on the right-hand side of Eq. (3) is negative, which contradicts the fact that the difference equals 1. Therefore, there is at least one k such that $j(k) \in \mathcal{A}_i$; by the pigeonhole principle, since there are exactly m basis vectors and exactly m disjoint sets $\mathcal{A}_1$, $\ldots$, $\mathcal{A}_m$, it follows that there is *exactly* one k such that $j(k) \in \mathcal{A}_i$. Since $j(1) < \ldots < j(m)$, it follows by induction that $j(i) \in \mathcal{A}_i$, as desired.

Therefore, we can rewrite Eq. (3) in the following form:

$$1 = x_{j(i)} - \gamma \sum_{k=1}^m x_{j(k)} p_{j(k),i}. \tag{4}$$

Again, since the subtrahend in Eq. (4) is positive, it follows immediately that

$$\begin{aligned} 1 &= x_{j(i)} - \gamma \sum_{k=1}^m x_{j(k)} p_{j(k),i} \\ &\le x_{j(i)}. \end{aligned}$$

Lastly, we note that by summing Eq. (4) over all $i = 1, \ldots, m$, we obtain

$$\begin{aligned}
m &= \sum_{i=1}^{m} 1 \\
&= \sum_{i=1}^{m} \left(x_{j(i)} - \gamma \left[\sum_{k=1}^{m} x_{j(k)} p_{j(k),i} \right] \right) \\
&= \left[\sum_{i=1}^{m} x_{j(1)} \right] - \gamma \left[\sum_{k=1}^{m} x_{j(k)} \left(\sum_{i=1}^{m} p_{j(k),i} \right) \right] \\
&= \left[\sum_{i=1}^{m} x_{j(1)} \right] - \gamma \left[\sum_{k=1}^{m} x_{j(k)} \right] \\
&= (1-\gamma) \sum_{i=1}^{m} x_{j(i)},
\end{aligned}$$

where we have used the fact that $\mathbf{p}_j$ is a probability vector, so $\mathbf{p}_j^\top \mathbf{1} = \sum_{i=1}^{m} p_{j,i} = 1$. Dividing through by $1 - \gamma$ gives that $\sum_{i=1}^{m} x_{j(i)} = \frac{m}{1-\gamma}$, as required. $\square$

Question 2 (Value Iteration Method)**.** *This is a first-order optimization method—starting with any vector* $\mathbf{y}^0$*, then iteratively update it*

$$y_i^{k+1} = \min_{j \in \mathcal{A}_i} c_j + \gamma \mathbf{p}_j^\top \mathbf{y}^k, \ \forall i. \tag{5}$$

Prove the contraction result:

$$\|\mathbf{y}^{k+1} - \mathbf{y}^*\|_\infty \leq \gamma \|\mathbf{y}^k - \mathbf{y}^*\|_\infty, \ \forall k.$$

where $\mathbf{y}^*$ *is the fixed-point or optimal value vector, that is,*

$$y_i^* = \min_{j \in \mathcal{A}_i} c_j + \gamma \mathbf{p}_j^\top \mathbf{y}^*, \ \forall i.$$

Answer. First, fix $1 \leq i \leq m$. First, let us suppose that $y_i^{k+1} \geq y_i^*$, so that $|y_i^{k+1} - y_i^*| = y_i^{k+1} - y_i^*$. Then, consider the fact that

$$\begin{aligned}
y^{k+1} - y_i^* &= \left(\min_{j \in \mathcal{A}_i} c_j + \gamma \mathbf{p}_j^\top \mathbf{y}^k \right) - y_i^* \\
&= \left(\min_{j \in \mathcal{A}_i} c_j + \gamma \mathbf{p}_j^\top \mathbf{y}^k \right) - \left(\min_{j \in \mathcal{A}_i} c_j + \gamma \mathbf{p}_j^\top \mathbf{y}^* \right)
\end{aligned}$$

Let the minimizing j in the minuend be j_0 and the minimizing j in the subtrahend be j_1. Then, we have that

$$c_{j_1} + \gamma \mathbf{p}_{j_1}^\top \mathbf{y}^k \geq c_{j_0} + \gamma \mathbf{p}_{j_0}^\top \mathbf{y}^k.$$

Combining these observations, we have that

$$\begin{aligned}
|y_i^{k+1} - y_i^*| &= y_i^{k+1} - y_i^* \\
&= (c_{j_0} + \gamma \mathbf{p}_{j_0}^\top \mathbf{y}^k) - (c_{j_1} + \gamma \mathbf{p}_{j_1}^\top \mathbf{y}^*) \\
&\le (c_{j_1} + \gamma \mathbf{p}_{j_1}^\top \mathbf{y}^k) - (c_{j_1} + \gamma \mathbf{p}_{j_1}^\top \mathbf{y}^*) \\
&= \gamma \mathbf{p}_{j_1}^\top (\mathbf{y}^k - \mathbf{y}^*) \\
&\le \gamma \|p_{j_1}\|_1 \cdot \|\mathbf{y}^k - \mathbf{y}^*\|_\infty \\
&= \gamma \|\mathbf{y}^k - \mathbf{y}^*\|_\infty,
\end{aligned}$$

where the final inequality follows by Hölder's inequality, and we have used the fact that $\|\mathbf{p}_j\|_1 = 1$ for all $1 \le j \le n$.

Next, let us suppose that $y_i^{k+1} \le y_i^*$, so that $|y_i^{k+1} - y_i^*| = y_i^* - y_i^{k+1}$. As before, let j_0 and j_1 be such that

$$c_{j_0} + \gamma \mathbf{p}_{j_0}^\top \mathbf{y}^k = \min_{j \in \mathcal{A}_i} c_j + \gamma \mathbf{p}_j^\top \mathbf{y}^k, \qquad c_{j_1} + \gamma \mathbf{p}_{j_1}^\top \mathbf{y}^* = \min_{j \in \mathcal{A}_i} c_j + \gamma \mathbf{p}_j^\top \mathbf{y}^*.$$

Note analogously

$$c_{j_0} + \gamma \mathbf{p}_{j_0}^\top \mathbf{y}^* \ge c_{j_1} + \gamma \mathbf{p}_{j_1}^\top \mathbf{y}^*.$$

Therefore, by virtually the same argument as before, we have that

$$\begin{aligned}
|y_i^{k+1} - y_i^*| &= y_i^* - y_i^{k+1} \\
&= (c_{j_1} + \gamma \mathbf{p}_{j_1}^\top \mathbf{y}^*) - (c_{j_0} + \gamma \mathbf{p}_{j_0}^\top \mathbf{y}^k) \\
&\le (c_{j_0} + \gamma \mathbf{p}_{j_0}^\top \mathbf{y}^*) - (c_{j_0} + \gamma \mathbf{p}_{j_0}^\top \mathbf{y}^k) \\
&= \gamma \mathbf{p}_{j_0}^\top (\mathbf{y}^* - \mathbf{y}^k) \\
&\le \gamma \|p_{j_1}\|_1 \cdot \|\mathbf{y}^k - \mathbf{y}^*\|_\infty \\
&= \gamma \|\mathbf{y}^k - \mathbf{y}^*\|_\infty.
\end{aligned}$$

Therefore, it is true unconditionally that $|y_i^{k+1} - y_i^*| \le \gamma \|\mathbf{y}^k - \mathbf{y}^*\|_\infty$. Since this is true for all i, it follows, taking the maximum of the left-hand side over all i, that $\|\mathbf{y}^{k+1} - \mathbf{y}^*\|_\infty \le \gamma \|\mathbf{y}^k - \mathbf{y}^*\|_\infty$, as desired. □

Question 3. *In the VI method, if starting with any vector $\mathbf{y}^0 \ge \mathbf{y}^*$ and assuming $\mathbf{y}^1 \le \mathbf{y}^0$, then prove the following entry-wise monotone property:*

$$\mathbf{y}^* \le \mathbf{y}^{k+1} \le \mathbf{y}^k, \ \forall k.$$

On the other hand, if we start from a vector such that

$$y_i^0 < \min_{j \in \mathcal{A}_i} c_j + \gamma \mathbf{p}_j^\top \mathbf{y}^0, \ \forall i$$

($\mathbf{y}^0$ in the interior of the feasible region), then prove the entry-wise monotone property:

$$\mathbf{y}^* \ge \mathbf{y}^{k+1} \ge \mathbf{y}^k, \ \forall k.$$

*This monotone property has been used in a recent paper (see [**SWWY17**]) on the VI method using samples.*

Answer. The proof is almost identical to the proof of Question 2. We begin, as before, by letting j_0 and j_1 be such that

$$c_{j_0} + \gamma \mathbf{p}_{j_0}^\top \mathbf{y}^k = \min_{j \in \mathcal{A}_i} c_j + \gamma \mathbf{p}_j^\top \mathbf{y}^k, \qquad c_{j_1} + \gamma \mathbf{p}_{j_1}^\top \mathbf{y}^* = \min_{j \in \mathcal{A}_i} c_j + \gamma \mathbf{p}_j^\top \mathbf{y}^*.$$

First, we consider the case where $\mathbf{y}^0 \geq \mathbf{y}^*$, and proceed by induction. In particular, we may assume that $\mathbf{y}^k \geq \mathbf{y}^*$. Then, note that

$$\begin{aligned} y_i^{k+1} - y_i^* &= (c_{j_0} + \gamma \mathbf{p}_{j_0}^\top \mathbf{y}^k) - (c_{j_1} + \gamma \mathbf{p}_{j_1}^\top \mathbf{y}^*) \\ &\geq (c_{j_0} + \gamma \mathbf{p}_{j_0}^\top \mathbf{y}^k) - (c_{j_0} + \gamma \mathbf{p}_{j_0}^\top \mathbf{y}^*) \\ &= \gamma \mathbf{p}_{j_0}^\top (\mathbf{y}^k - \mathbf{y}^*) \\ &\geq 0. \end{aligned}$$

Here, the final inequality follows from the fact that $\mathbf{y}^k - \mathbf{y}^* \geq \mathbf{0}$ and $\mathbf{p}_{j_0} \geq \mathbf{0}$, and so $\mathbf{p}_{j_0}^\top (\mathbf{y}^k - \mathbf{y}^*) \geq 0$. Since this holds for all i, it follows that $\mathbf{y}^{k+1} \geq \mathbf{y}^*$.

To see that $\mathbf{y}^{k+1} \leq \mathbf{y}^k$, note that by the inductive hypothesis, $\mathbf{y}^k \leq \mathbf{y}^{k-1}$. Let $j_2 \in \mathcal{A}_i$ be such that

$$\mathbf{y}_i^k = c_{j_2} + \gamma \mathbf{p}_{j_2}^\top \mathbf{y}^{k-1}.$$

Then, it follows that

$$\begin{aligned} \mathbf{y}_i^{k+1} &= \min_{j \in \mathcal{A}_i} c_j + \gamma \mathbf{p}_j^\top \mathbf{y}^k \\ &\leq c_{j_2} + \gamma \mathbf{p}_{j_2}^\top \mathbf{y}^k \\ &= c_{j_2} + \gamma \mathbf{p}_{j_2}^\top ([\mathbf{y}^k - \mathbf{y}^{k+1}] + \mathbf{y}^{k+1}) \\ &= \gamma \mathbf{p}_{j_2}^\top (\mathbf{y}^k - \mathbf{y}^{k+1}) + c_{j_2} + \gamma \mathbf{p}_{j_2}^\top \mathbf{y}^{k-1} \\ &= \gamma \mathbf{p}_{j_2}^\top (\mathbf{y}^k - \mathbf{y}^{k-1}) + y_i^k. \end{aligned}$$

Now, since $\gamma > 0$, and $\mathbf{p}_{j_2} \geq \mathbf{0}$, and $\mathbf{y}^k - \mathbf{y}^{k-1} \leq \mathbf{0}$ by the inductive hypothesis, it follows that $\gamma \mathbf{p}_{j_2}^\top (\mathbf{y}^k - \mathbf{y}^{k-1}) < 0$, and so $\mathbf{y}_i^{k+1} \leq \mathbf{y}^k$. Since this holds for all i, it follows that $\mathbf{y}^{k+1} \leq \mathbf{y}^k$, as desired.

Next, we consider the case where $\mathbf{y}^0 \leq \mathbf{y}^*$. Again, we may assume by induction that $\mathbf{y}^{k-1} \leq \mathbf{y}^k \leq \mathbf{y}^*$. Letting j_0 and j_1 be as before, we have that

$$\begin{aligned} y_i^{k+1} - y_i^* &= (c_{j_0} + \gamma \mathbf{p}_{j_0}^\top \mathbf{y}^k) - (c_{j_1} + \gamma \mathbf{p}_{j_1}^\top \mathbf{y}^*) \\ &\leq (c_{j_1} + \gamma \mathbf{p}_{j_1}^\top \mathbf{y}^k) - (c_{j_1} + \gamma \mathbf{p}_{j_1}^\top \mathbf{y}^*) \\ &= \gamma \mathbf{p}_{j_1}^\top (\mathbf{y}^k - \mathbf{y}^*) \\ &\leq 0, \end{aligned}$$

where the final inequality follows from the fact that $\mathbf{y}^k - \mathbf{y}^* \leq \mathbf{0}$. Since this holds for all i, $\mathbf{y}^{k+1} \leq \mathbf{y}^*$. Likewise,

$$\begin{aligned} \mathbf{y}_i^{k+1} &= c_{j_0} + \gamma \mathbf{p}_{j_0}^\top \mathbf{y}^k \\ &= c_{j_0} + \gamma \mathbf{p}_{j_0}^\top ([\mathbf{y}^k - \mathbf{y}^{k+1}] + \mathbf{y}^{k+1}) \\ &= \gamma \mathbf{p}_{j_0}^\top (\mathbf{y}^k - \mathbf{y}^{k+1}) + c_{j_0} + \gamma \mathbf{p}_{j_2}^\top \mathbf{y}^{k-1} \\ &\leq \gamma \mathbf{p}_{j_2}^\top (\mathbf{y}^k - \mathbf{y}^{k-1}) + y_i^k, \end{aligned}$$

where the final inequality follows from the fact that

$$y_i^k = \min_{j \in \mathcal{A}_i} c_j + \gamma \mathbf{p}_j^\top \mathbf{y}^{k-1} \le c_{j_0} + \gamma \mathbf{p}_{j_0}^\top \mathbf{y}^{k-1}.$$

Moreover, again since $\gamma > 0$, and $\mathbf{p}_{j_2} \ge \mathbf{0}$, and $\mathbf{y}^k - \mathbf{y}^{k-1} \ge \mathbf{0}$ by the inductive hypothesis, it follows that $\gamma \mathbf{p}_{j_2}^\top (\mathbf{y}^k - \mathbf{y}^{k-1}) > 0$. Therefore $y_i^{k+1} \ge y_i^k$ for all i, and so $\mathbf{y}^{k+1} \ge \mathbf{y}^k$. This completes the proof. □

Question 4. *Rather than go through all state values in each iteration, we modify the VI method, calling it "RamdomVI": In the k-th iteration, randomly select a subset of states B^k and do*

$$y_i^{k+1} = \min_{j \in \mathcal{A}_i} c_j + \gamma \mathbf{p}_j^T \mathbf{y}^k, \ \forall i \in B^k. \tag{6}$$

In RandomVI, we only update a subset of state values at random in each iteration.

What can you tell the convergence of the RandomVI method? Does it make a difference with the classical VI method? How is the sample size affect the performance? Use simulated computational experiments to verify your claims.

Rather than randomly select a subset of all states in each iteration, suppose we build an "influence tree" from a given subset of states, say B, for all sates, denoted by $I(B)$, that are connected by any state in B. Then when states in B are updated in the current iteration, then selected a subset of states in $I(B)$ for updating in the next iteration. Redo the computational experiments using this strategy for a sparsely connected ($\mathbf{p}_j$ is a very sparse distribution vector for each action j) MDP network. In doing so, many unimportant or irrelevant states may be avoided which results a state-reduction.

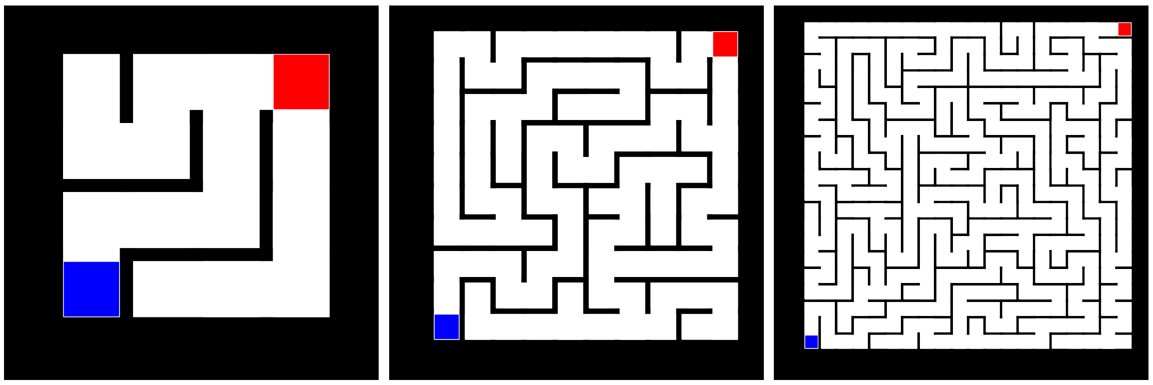
Note on Code. The code used to generate MDPs can be found in `util.jl`. The code used to solve MDPs can be found in `vi.jl`. The code used to perform numerical experiments can be found in `expr.jl`. The code used to generate plots and perform analysis can be found in `plots.Rmd`. All simulations shown in the plots were conducted with $\gamma = 0.9$ and $\mathbf{y}^0 = \vec{\mathbf{1}}$, unless otherwise indicated.

Answer. For this and all subsequent questions, we perform our numerical experiments on two kinds of simulated two-dimensional maze games.

The first, which we call the "standard maze game," consists of an $n \times n$ two-dimensional maze. Each state represents a position in the maze. We imagine that the player can move from state to state (that is, up, down, left, or right) as allowed by the constraints of the maze. We further require that each action incurs an immediate cost of 1, while the player can obtain a unit reward (i.e., a cost of -1) by reaching the final square of the maze, which by convention we place at position (n, n).

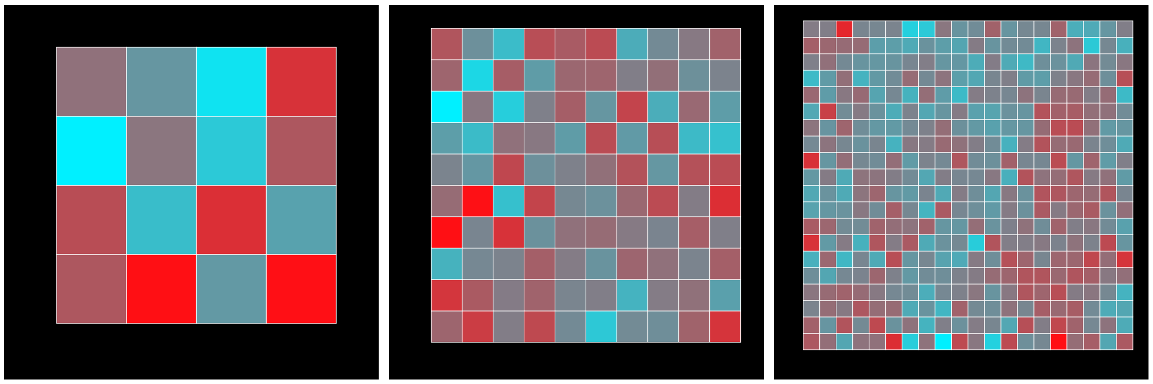
Example standard mazes of side-length 4, 10, and 20 (i.e., sizes 16, 100, and 400) are shown in Figure 1 below.

In addition to the "standard maze game," we also test our optimization methods on a second two-dimensional maze game, which we call the "terrain maze game." As in the "standard maze game," each state in the terrain maze game represents a position in the maze. As before, we imagine that the player can move from state to state only by travelling a single unit up, down, left, or right. (The player is only constrained from moving off the



(a) $n = 4$ (b) $n = 10$ (c) $n = 20$

Figure 1: Three example "standard mazes." Players begin in the bottom left corner on the blue tile (i.e., position $(1, 1)$) and proceed to the red tile (i.e., position (n, n)).



(a) $n = 4$ (b) $n = 10$ (c) $n = 20$

Figure 2: Three example "terrain mazes." Players begin in the bottom left corner on (i.e., position $(1, 1)$) and proceed to the top right corner (i.e., position (n, n)). Greater costs are incurred for moving uphill than downhill. High positions are indicated by red colors, and low positions are indicated by blue colors.

maze entirely.) The player receives a reward (i.e., a cost of -1) by reaching the final square of the maze, which again we place at position (n, n). However, in contrast to the standard maze game, the player's movements incur different costs at different positions. In particular, we create a height function $H : \{1, \ldots, n\} \times \{1, \ldots, n\} \to \mathbb{R}$.[2] Then, the cost of movement is set to be the difference in heights between the player's destination position and their current position, normalized appropriately.

Example terrain mazes of side-length 4, 10, and 20 (i.e., sizes 16, 100, and 400) are shown in Figure 2 below.

The terrain mazes and standard mazes have very different properties. In both cases, the actions available at any state correspond to a very sparse transition probability vectors, since players are constrained to move along cardinal directions at a rate of a single unit.

[2]More specifically, this is created by (i.i.d.) drawing a value $X_{i,j} \sim \mathcal{N}(0, 10)$ for each position (i, j); then, the value $H_{i,j}$ is set equal to the average of $X_{i',j'}$ for all positions (i', j') immediately neighboring (i, j).

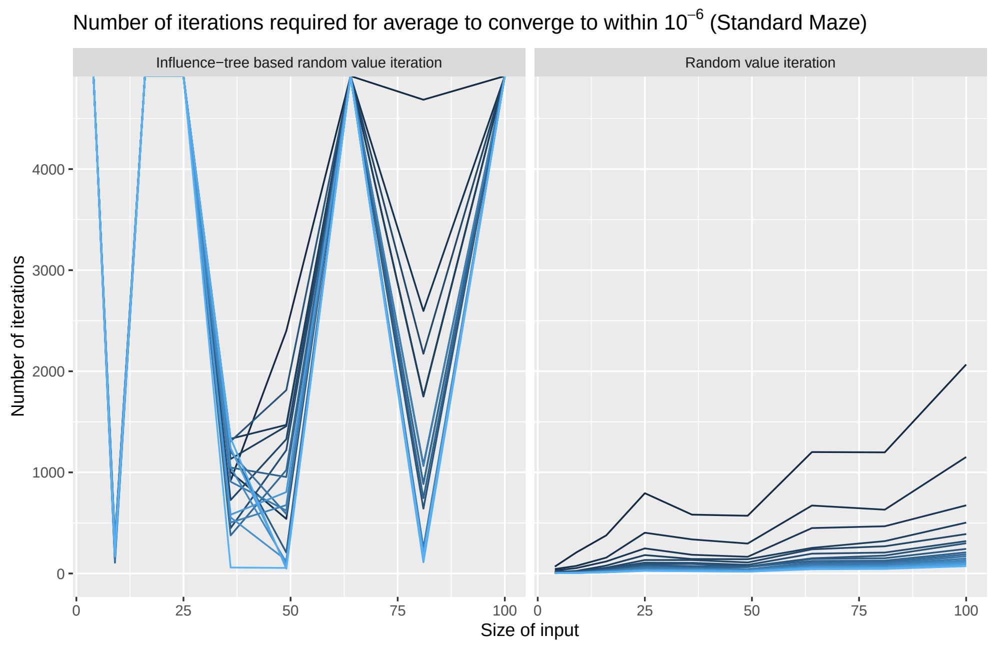
However, in the standard maze game, the cost-to-go at any position is extremely sensitive to the costs-to-go at states that are very distant. In a 10×10 standard maze, the initial tile (i.e., position $(1, 1)$) is often between 25 and 30 units away from the destination tile (i.e., position $(10, 10)$). However, the cost-to-go is essentially a function of that distance, and so accurately updating the cost-to-go of the initial tile depends on accurately updating the costs-to-go at all of the 25 to 30 tiles between it and the destination tile. In contrast, in the terrain maze game, the cost-to-go is much less sensitive to far away positions, because local immediate costs, dictated by the slopes one must climb or go down to move locally, are more significant.

In particular, neither strategy outperformed classical iteration in our simulations across input sizes and parameter values. In the case of the standard maze, the influence-tree–based random value iteration actually failed catastrophically on many inputs for reasons detailed below. See Figures 5a and 5b.

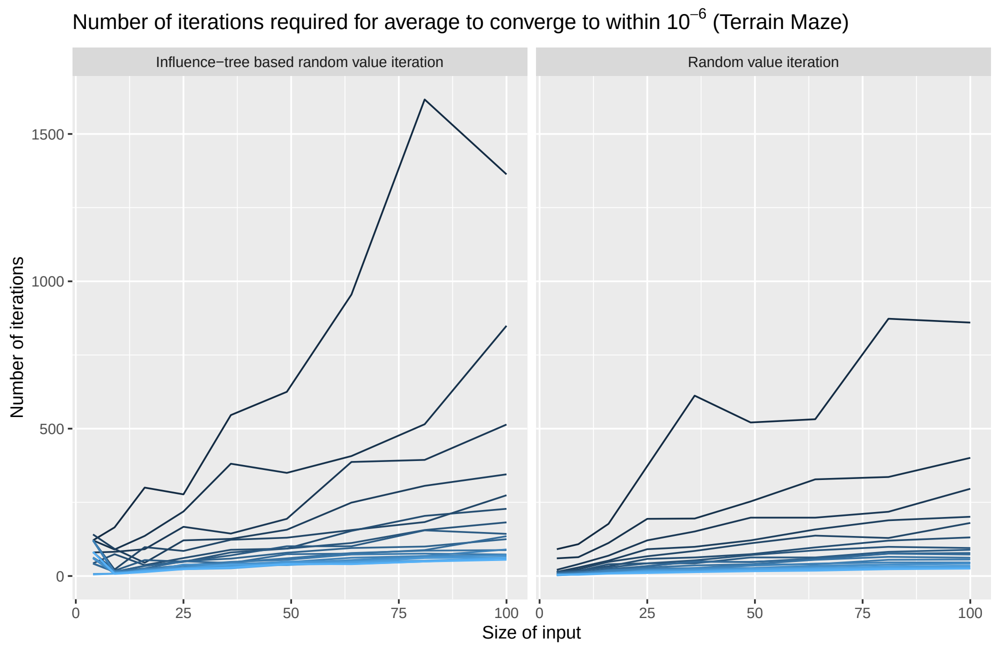
This difference can be seen in the different convergence properties of the different strategies, as can be seen by comparing Figures 3a and 3b. In particular, while the number of iterations required to reach convergence (operationalized as the average norm of solution being within 10^{-6} of the true solution) is lower across the board in the terrain maze, the influence-tree method effectively does not converge. There is a dependence on the probability of inclusion p, the parameter which governs how large the set of states we update from at each iteration is, but if we choose p to be approximately $\frac{1}{n^2}$, we see that the influence-tree–based method corresponds, in effect, to updating the value according to a random walk on the maze. In particular, since the value at $(1, 1)$ can only be accurately updated after the random walk has visited the states between (n, n) and $(1, 1)$ a sufficient number of times, it can require a very large number of iterations before the value at the initial tile $(1, 1)$ or other hard-to-reach tiles is updated. Since the fully random strategy does not have this property—that is, every position has an equal chance of being updated at every step—it is not a sensitive to such pathological behavior and "unlucky" random walks. However, even in this latter case, the influence-tree–based random value iteration strategy does not actually outperform random value iteration.

A second point worth addressing is that, when using "random value iteration" and "influence-tree–based random value iteration," the speed of convergence is not necessarily well-benchmarked by the number of iterations. In particular, if exactly half of the states are chosen at each iteration in RandomVI, then each update iteration is approximately half as expensive as in ClassicalVI.[3] Therefore, the proper metric is the number of iterations required to reach convergence times the parameter p, which controls the number of updates performed at each iteration in both methods. The results are shown in Figures 4a and 4b. Due to the pathological behavior noted before, we focus on Figure 4b. Here we can see that a larger probability of inclusion p (i.e., updating more states at each iteration) leads to modestly faster convergence times, consistent with our intuition about the dependence structure of values required for convergence. (I.e., we may not be able to accurately update

[3]In fact, this is not quite true: there is additional overhead required to manage the memory where $\mathbf{y}^k$ and $\mathbf{y}^{k+1}$ are stored in comparison with alternative methods, where $\mathbf{y}^k$ and $\mathbf{y}^{k+1}$ can actually simply be switched at the end of each iteration; additionally, in the case of influence-tree–based random value iteration, additional memory is required to calculate which states are reachable from the states currently being explored.



(a) Standard maze.



(b) Terrain maze.

Figure 3: This plot shows the number of iterations required for (1) influence-tree–based random value iteration and (2) random value iteration to converge to within 10^{-6} in the ℓ^2-norm to the true solution on standard mazes of total size (i.e., number of states m equal to $n \times n$) as indicated along the x-axis. Darker colored lines indicate smaller values of the p parameter (and consequently slower convergence) while lighter colored lines indicate larger values of the p parameter (and consequently faster convergence).

the value of one state until states far away from it have been updated; hence, randomly updating small subsets—and consequently having a large variance in the number of times any individual state is updated—leads to slower convergence.)

Additional figures showing the convergence rate under various conditions for these two randomized strategies, viz., Figures 7, 8, 9, and 9, are included in the Appendix. Solutions to the mazes output by the various algorithms are also included in the Appendix. □

Question 5. *Here is another modification, called CyclicVI: In the k-th iteration do:*

- *Initialize $\tilde{\mathbf{y}}^k = \mathbf{y}^k$.*
- *For $i = 1$ to m*
$$\tilde{y}_i^k = \min_{j \in \mathcal{A}_i} c_j + \gamma \mathbf{p}_j^T \tilde{\mathbf{y}}^k \tag{7}$$
- *$\mathbf{y}^{k+1} = \tilde{\mathbf{y}}^k$.*

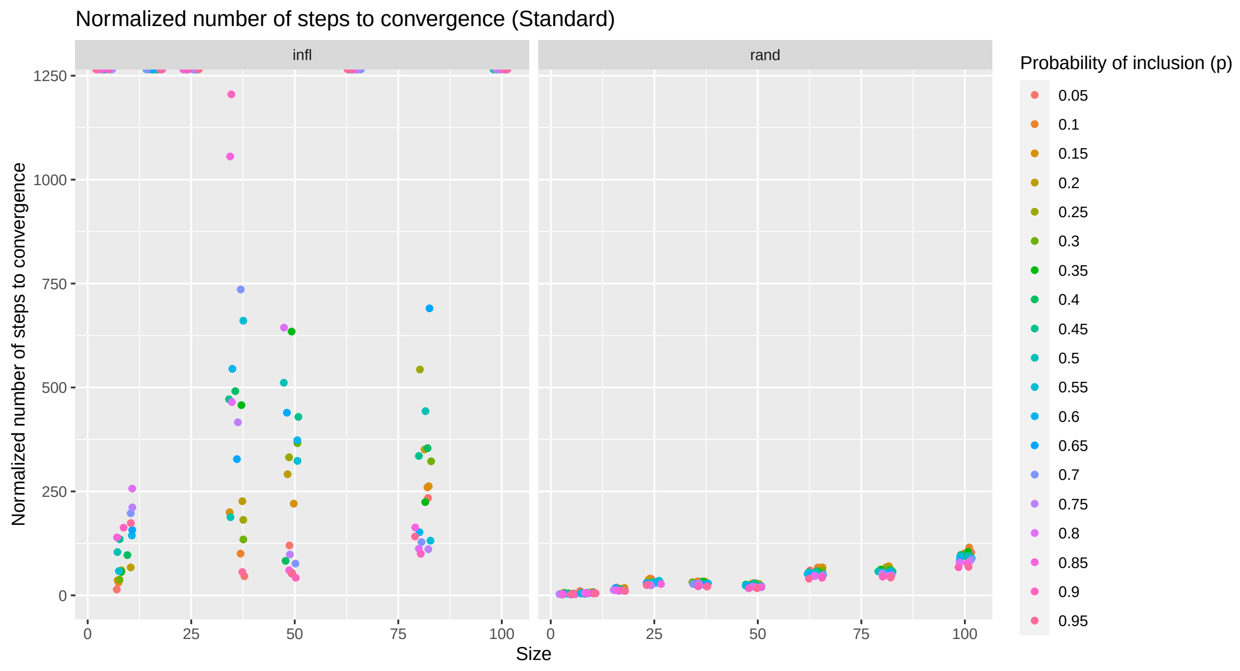
In the CyclicVI method, as soon as a state value is updated, we use it to update the rest of state values.

What can you tell the convergence of the CyclicVI method? Does it make a difference with other VI methods? Use simulated computational experiments to verify your claims. How is this cyclic method related to the method at the bottom of Question 4?

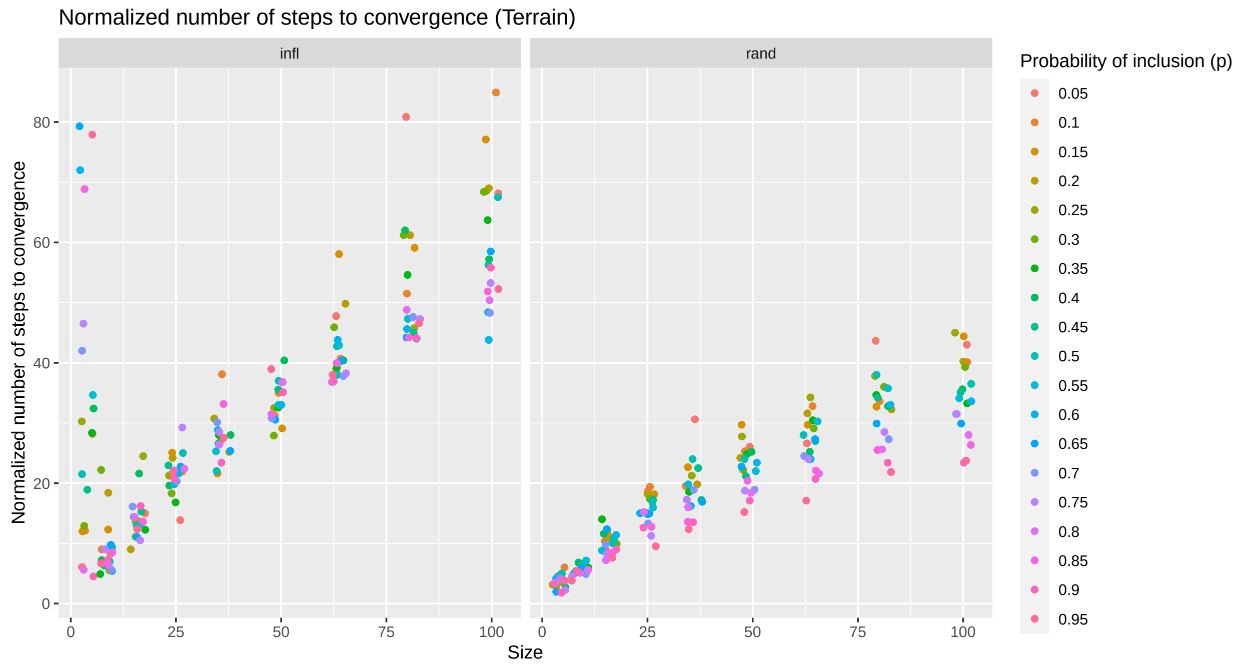
Answer. The results of our simulation on all methods (including CyclicVI) are shown in Figures 5a and 5b, and 6a and 6b below.

As we can see, cyclic value iteration converges at roughly the same rate as classical value iteraion for the terrain maze, and faster for the standard maze. (Approximately twice as fast.)

In particular, it is not hard to see that the rate of convergence of cyclic value iteration is bounded above by the rate of convergence of value iteration. The pointwise bound given in Question 2, viz.,
$$|y_i^{k+1} - y_i^*| \le \gamma \|\mathbf{y}^k - \mathbf{y}^*\|_\infty \tag{8}$$
carries over exactly, i.e., the same proof gives that
$$|\tilde{y}_i^{k+1} - y_i^*| \le \gamma \|\tilde{\mathbf{y}}^k - \mathbf{y}^*\|_\infty \tag{9}$$
Therefore, with each update of $\tilde{\mathbf{y}}^k$ in cyclic iteration, the vector $\tilde{\mathbf{y}}^k$ gets slightly closer to $\mathbf{y}^*$. However, since the bound is in terms of the ℓ^∞ norm, and each update occurs along an independent axis, the bound does not guarantee any additional speedup over classical value iteration. In particular, if $|\tilde{y}_m^k - y_m^*| = \|\tilde{\mathbf{y}}^k - \mathbf{y}^*\|_\infty$, then the Inequality (9) will exactly match the inequality given in Question 2. However, if the coordinates which differ more occur earlier, e.g.,
$$|\tilde{y}_1^k - y_1^*| = \|\tilde{\mathbf{y}}^k - \mathbf{y}^*\|_\infty \qquad |\tilde{y}_j^k - y_j^*| < \|\tilde{\mathbf{y}}^k - \mathbf{y}^*\|_\infty,\ \forall j > 1$$
then the Inequality (9) can be improved by a factor of at least γ over the guarantee given in Question 2.

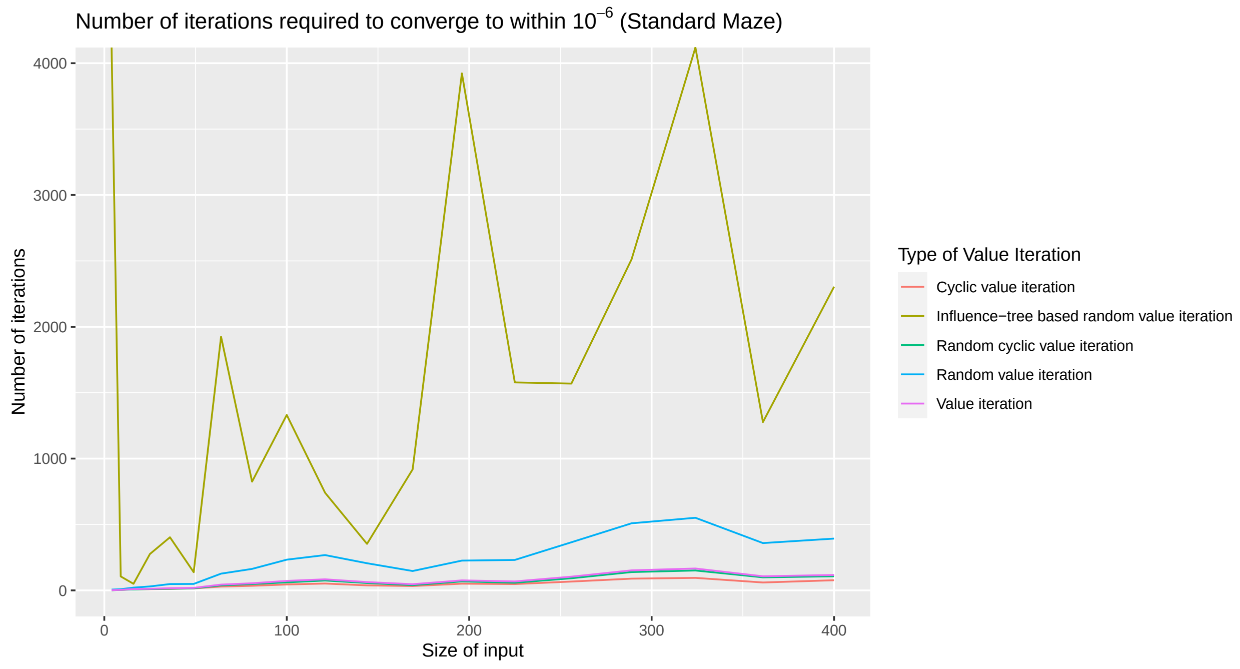


(a) Standard maze.

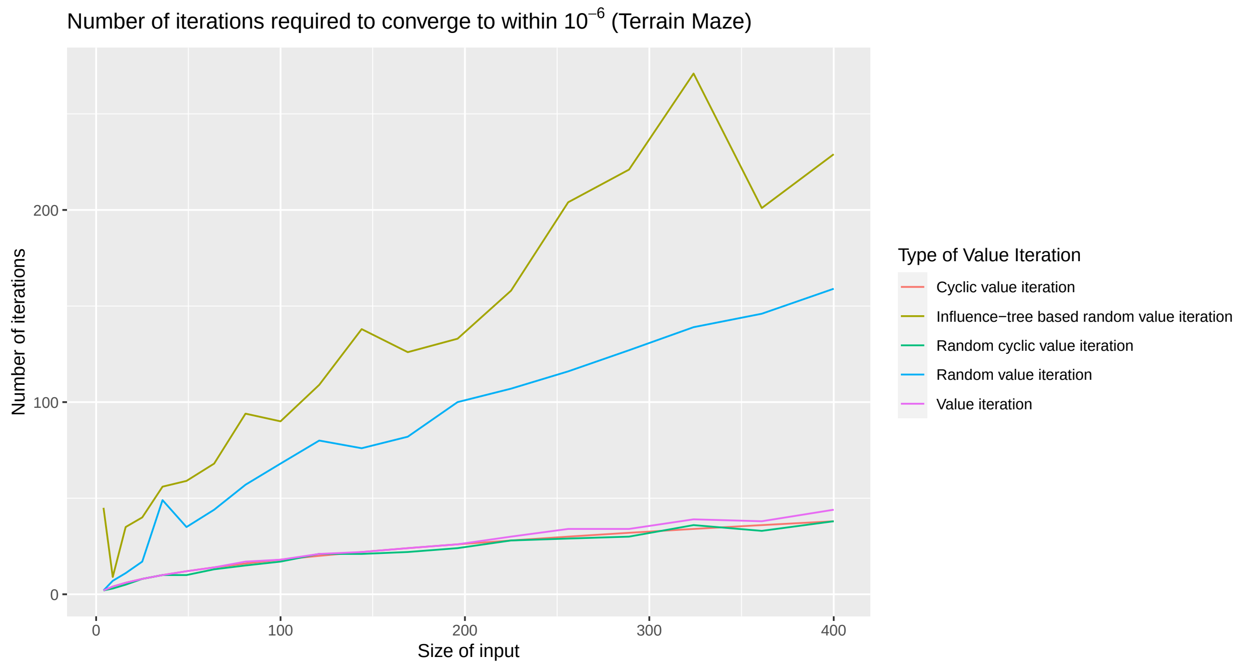


(b) Terrain maze.

Figure 4: The normalized number of steps required for the average norm of the difference between the current values $\mathbf{y}^k$ and the true values $\mathbf{y}^*$ to fall below 10^{-6}. (That is, $p \cdot k_{\max}$, where $k_{\max}$ is the smallest index such that the average of the norms of the differences in every simulation to be below 10^{-6}.) We see that larger values of p—which are closer to classical value iteration—show modest advantages in the total number of operations required (and hence wall-clock speed of convergence). The pathological failure of influence-tree–based random value iteration is shown in the top left panel.

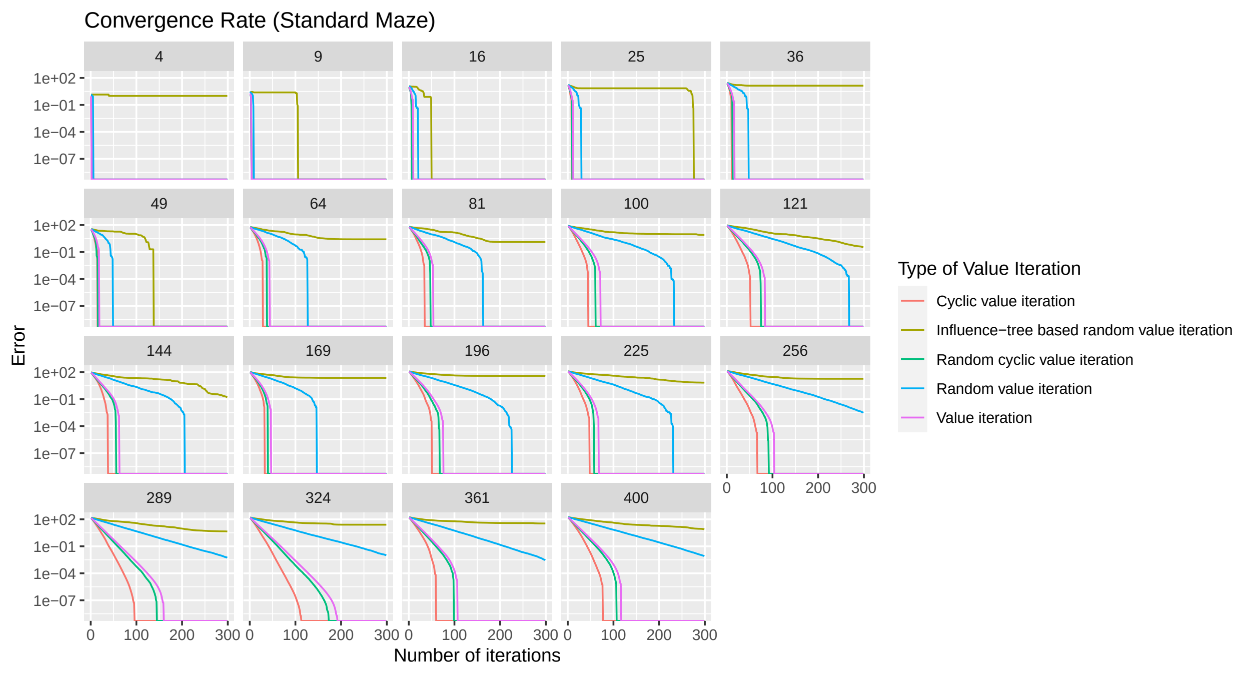


(a) Standard maze.

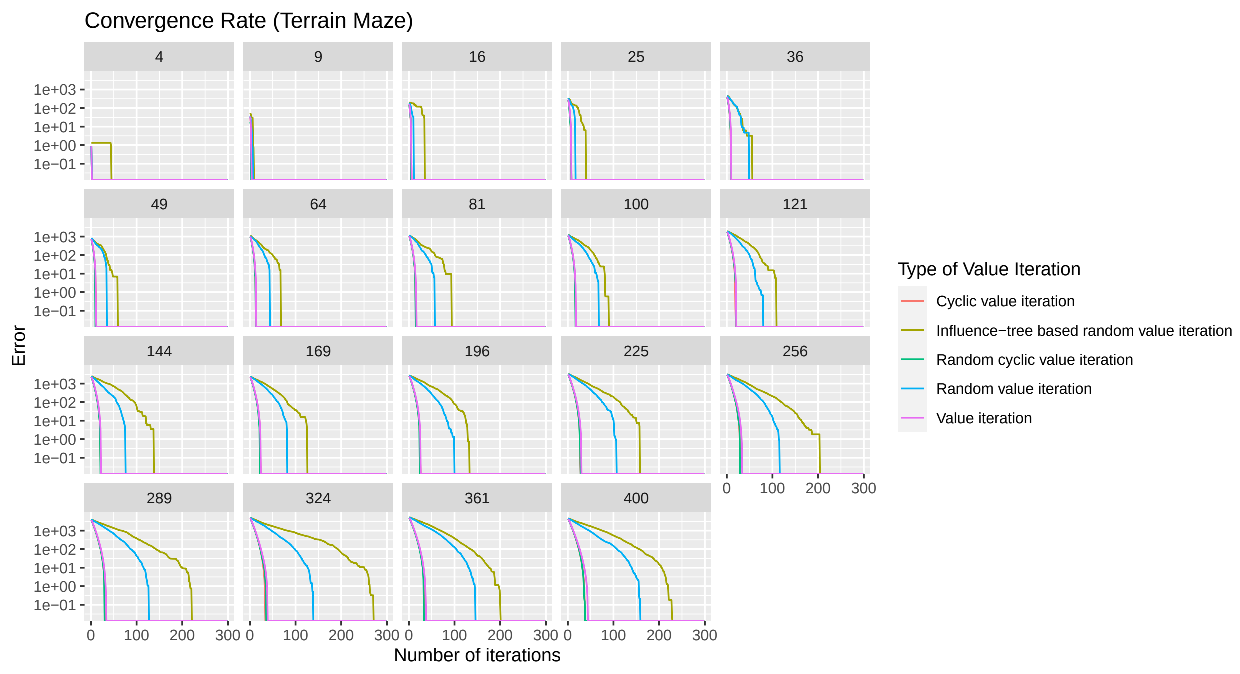


(b) Terrain maze.

Figure 5: Number of iterations required to reach convergence (within 10^{-6} of the true solution in the ℓ^2 norm) on a variety of input sizes. Here $\gamma = 0.9$ for all methods and $p = 0.5$ for RandomVI and variations.



(a) Standard maze.



(b) Terrain maze.

Figure 6: Convergence rates of various value iteration methods. Here $\gamma = 0.9$ for all methods and $p = 0.5$ for RandomVI and variations.

In the extreme case, if the value of $i \in \{1, \dots, m\}$ for which $|\tilde{y}_i^k - y_i^*|$ is maximal is, at inner iteration i, precisely i, then cyclic iteration will result in a substantial speedup over classical value iteration. In practice, we see a modest speedup over classical value iteration in our maze problems.

The shared intuition behind cyclical iteration and influence-tree–based value iteration can therefore be made clear in the following way: suppose that for many pairs of states $i, j \in \{1, \dots, m\}$ where j can be reached from i satisfy $i < j$. Then, if we set $B = \{1\}$, the chain of influenced states

$$I = \{1\} \to B(I) = B(\{1\}) \to \cdots \to B(\cdots(B(I))) = B(\cdots(B(\{1\})))$$

will look approximately like

$$\{1\} \to \{2\} \to \cdots \to \{m\}.$$

Then, influence-tree–based value iteration and cyclic value iteration will have very similar algorithmic traces. If the two chains are equal, the two algorithmic traces will in fact be identical (although our counting of iterations—i.e., outer iterations, in the case of cyclic iteration—does not immediately make this obvious).

The advantage of cyclic iteration, however, is that every state is guaranteed to be updated at least once after the completion of each outer iteration. Therefore, it has superior convergence properties, since it has better sampling behavior. In particular, if a particular policy gives rise to a non-recurrent markov chain, as is the case with our standard maze, it is possible that influence-tree–based value iteration may fail to update the values at certain states appropriately, wherease with cyclic iteration, this cannot arise. □

Question 6. *In the CyclicVI method, rather than with the fixed cycle order from 1 to m, we follow a random permutation order, or sample without replacement to update the state values. More precisely, in the k-th iteration do:*

0. Initialize $\tilde{\mathbf{y}}^k = \mathbf{y}^k$ *and* $B^k = \{1, 2, \dots, m\}$

1. – *Randomly select* $i \in B^k$

– *Set*

$$\tilde{\mathbf{y}}_i^k = \min_{j \in \mathcal{A}_i} c_j + \gamma \mathbf{p}_j^\top \tilde{\mathbf{y}}^k \tag{10}$$

– *Remove* i *from* B^k *and return to Step 1.*

3. Set $\mathbf{y}^{k+1} = \tilde{\mathbf{y}}^k$.

We call it the randomly permuted CyclicVI or RPCyclicVI in short

What can you tell the convergence of the RPCyclicVI method? Does it compare with other VI methods? Use simulated computational experiments to verify your claims.

Answer. As before, the results of our simulation on all methods (including CyclicVI) are shown in Figures 5a and 5b, and 6a and 6b. We see that for large terrain maze problems, cyclic value iteration and randomly permuted cyclic value iteration modestly outperform value iteration. For our standard maze problems, randomly permuted cyclic value iteration modestly outperforms standard value iteration, but does not outperform cyclic value iteration.

We begin by noting that, by the same argument as in Question 5, random permutation cyclic value iteration should not be slower than classical value iteration.

Moreover, we should, in general, expect some speedup over classical value iteration. The reason for this is straightforward. Consider again the influence chain exhibited in Question 5:

$$I = \{1\} \to B(I) = \{2\} \to \cdots \to B(\cdots(B(I))) = \{m\}.$$

In general, except in contrived problems, if there is a cyclic (or approximately cyclic) dependence structure among the various states, there is no reason to expect that it is in the order the states are arbitrarily laid out in the statement of the problem. It is just as reasonable to assume that

$$I = \{\pi(1)\} \to B(I) = \{\pi(2)\} \to \cdots \to B(\cdots(B(I))) = \{\pi(m)\}$$

for some permutation $\pi : \{1, \ldots, m\} \to \{1, \ldots, m\}$. In particular, if the true permutation is very different from $\pi_{\text{cyclic}} : i \mapsto (i+1) \pmod{m}$—which is, in effect, the only permutation considered by cyclic value iteration—we will never get the speedup associated with sequentially updating according to the true influence chain. Thus, while in the worst-case scenario, we cannot improve our bound for cyclic iteration over classical value iteration, on average, random permutation cyclic value iteration should, on average, offer some speedup.

This tradeoff can be phrased in terms of variance: if the *true* dependence chain looks very similar to π_{cyclic}, then we can expect cyclic value iteration to outperform random permutation cyclic value iteration. However, in cases where the true dependence chain looks very different from π_{cyclic}, we can expect random permutation cyclic value iteration to occasionally take advantage of the true dependence chain, since it updates according to a new random dependence chain on each iteration. Cyclic permutation, on the other hand, never changes the order in which it performs updates, i.e., never considers an alternative dependence chain. As a result, we should expect to see some speedup in these cases.

This intuition is largely what we see in our standard maze examples. Because the starting tile is $(1, 1)$ and the destination tile is (n, n), heuristically, a policy that solves the maze will tend to direct the player toward positions (i, j) for larger and and larger i and j. Since, in our implementation of cyclic value iteration, states are updated in dictionary order of their corresponding position, this means that π_{cyclic} is reasonably close to the true dependence chain, and so cyclic value iteration outperforms random permutation cyclic value iteration. □

References

[LY16] David G Luenberger and Yinyu Ye. *Linear and nonlinear programming*. 4th ed. Vol. 2. Springer, Cham, 2016. DOI: https://doi.org/10.1007/978-3-319-18842-3.

A Additional Figures

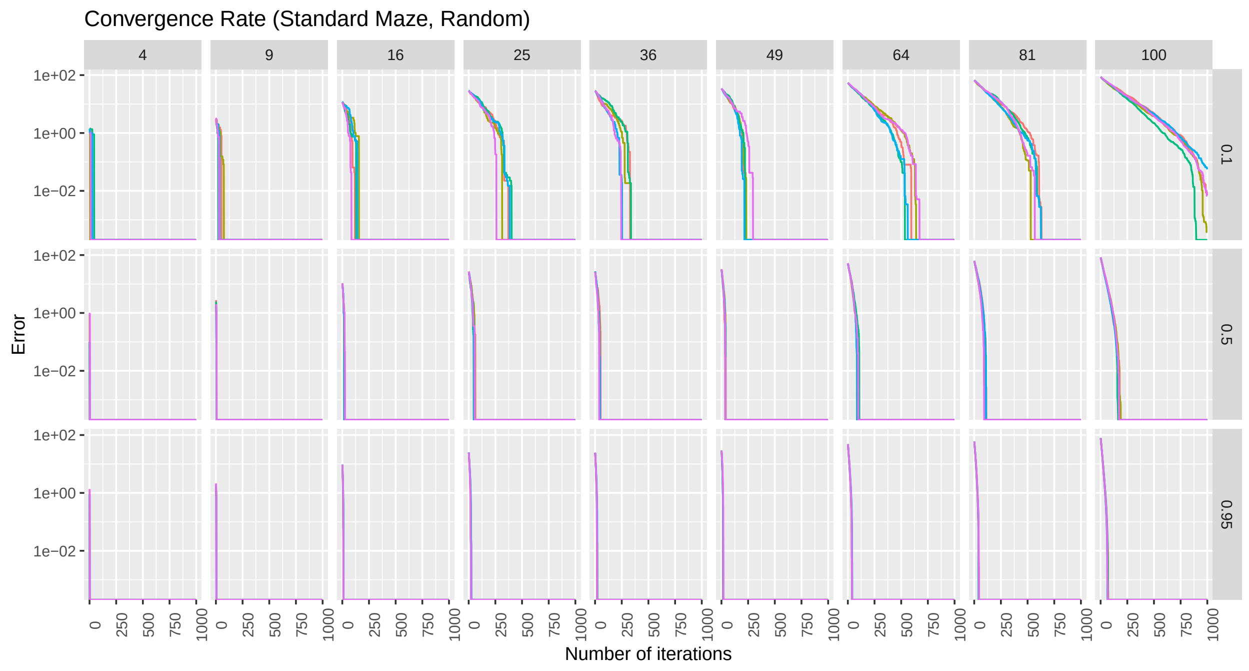


Figure 7: Convergence rate of random value iteration on the standard maze for different maze sizes and values of p. Distinct lines represent distinct simulations.

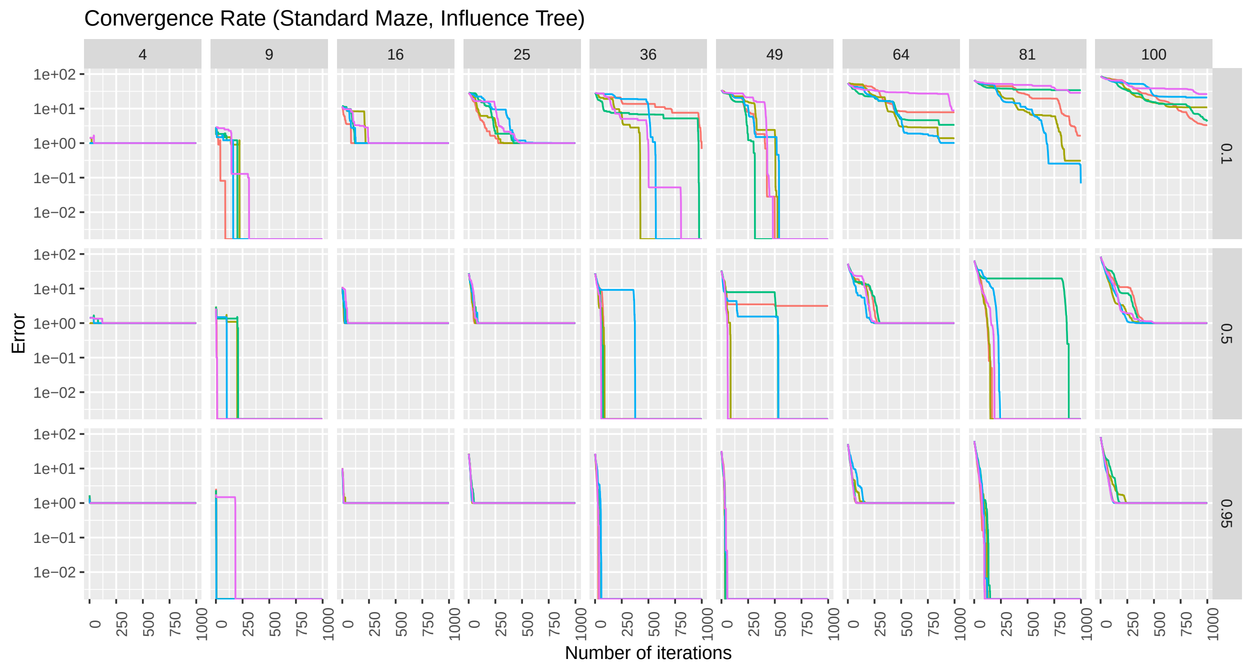


Figure 8: Convergence rate of influence-tree–based random value iteration on the standard maze for different maze sizes and values of p. Distinct lines represent distinct simulations.

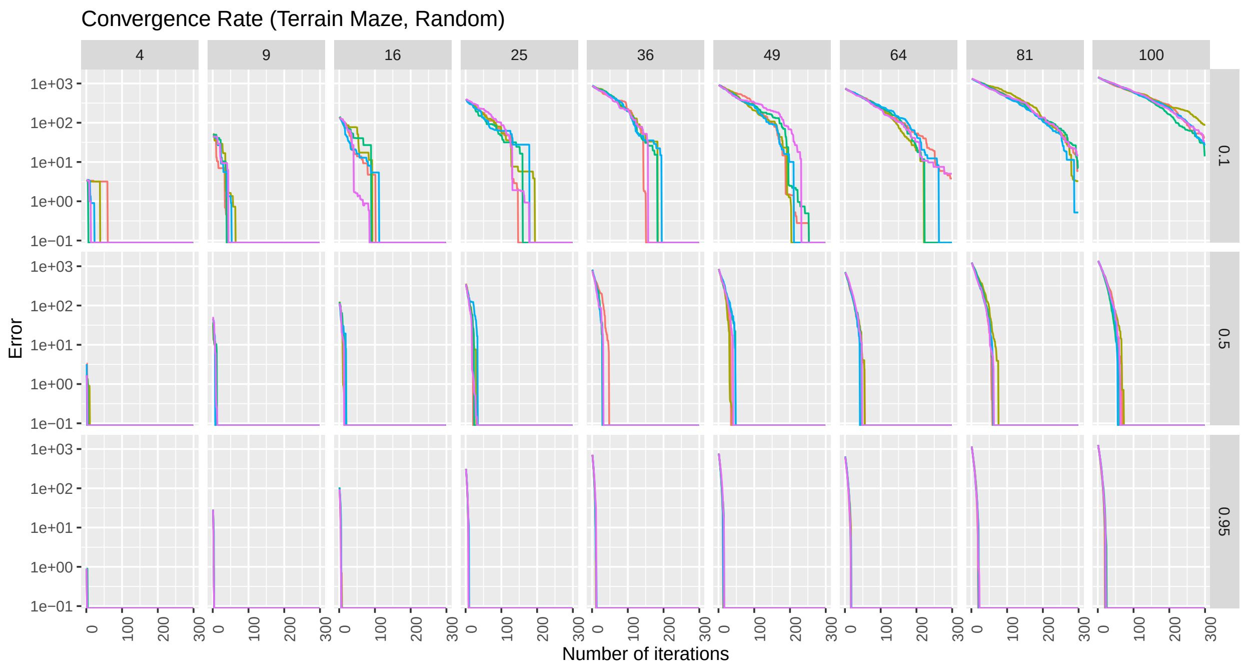


Figure 9: Convergence rate of random value iteration on the terrain maze for different maze sizes and values of p. Distinct lines represent distinct simulations.

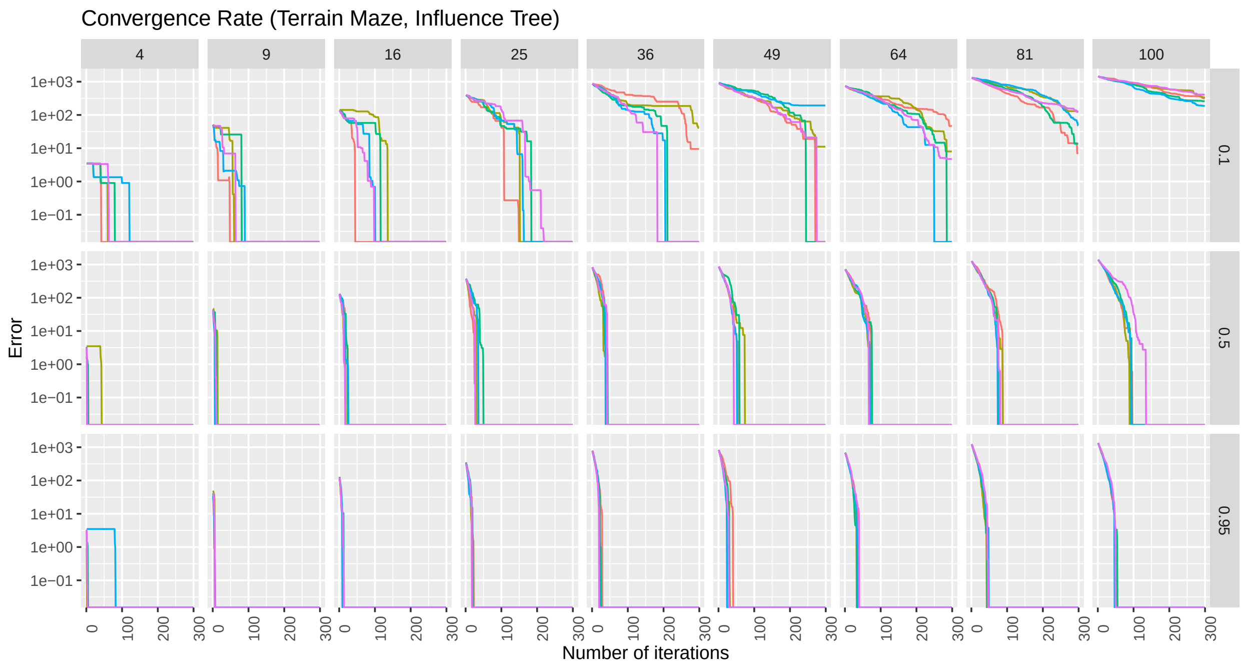


Figure 10: Convergence rate of influence-tree–based random value iteration on the terrain maze for different maze sizes and values of p. Distinct lines represent distinct simulations.

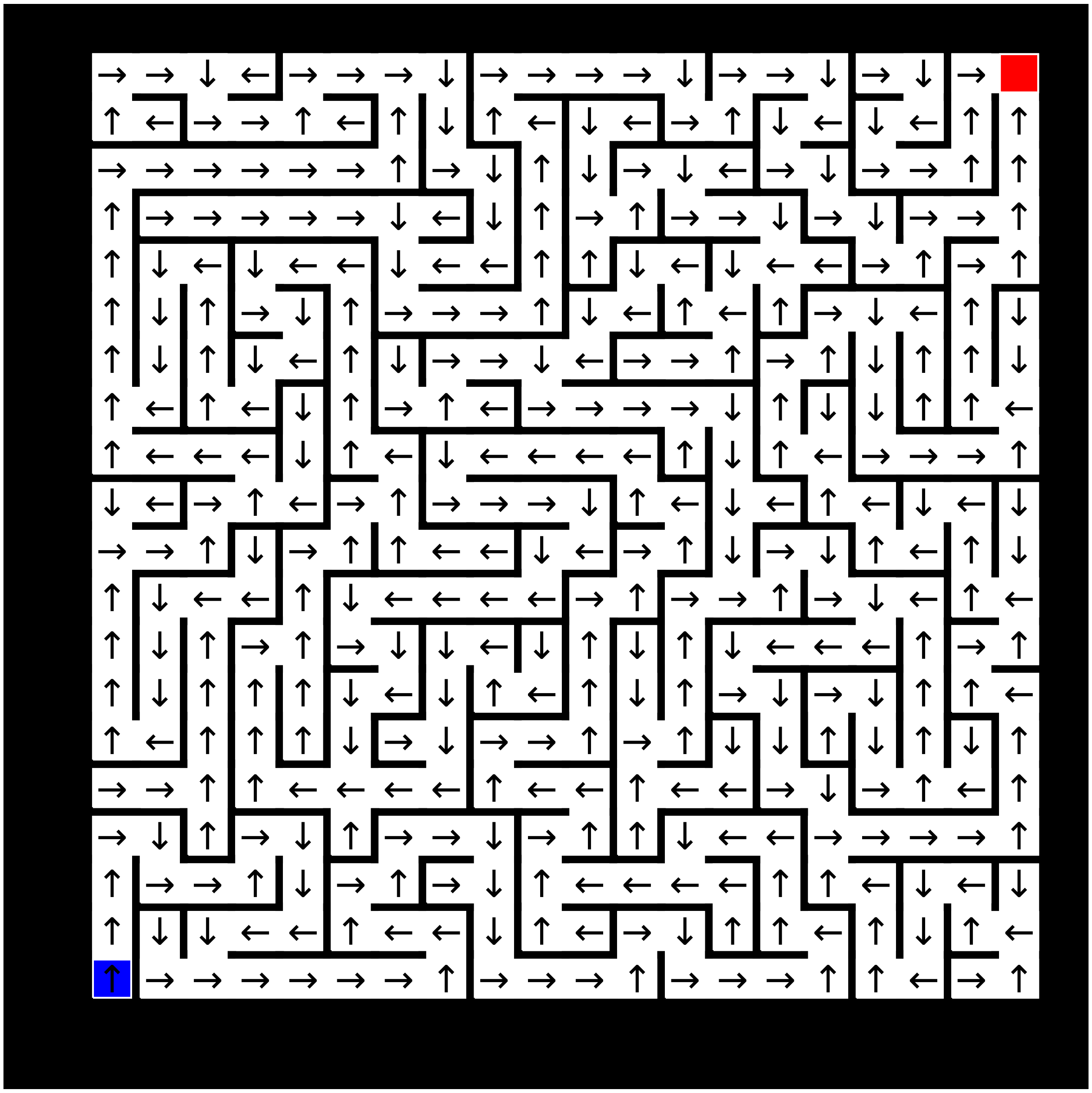


Figure 11: The solution to a 20 × 20 standard maze, using cylic iteration and $\gamma = 0.99$.

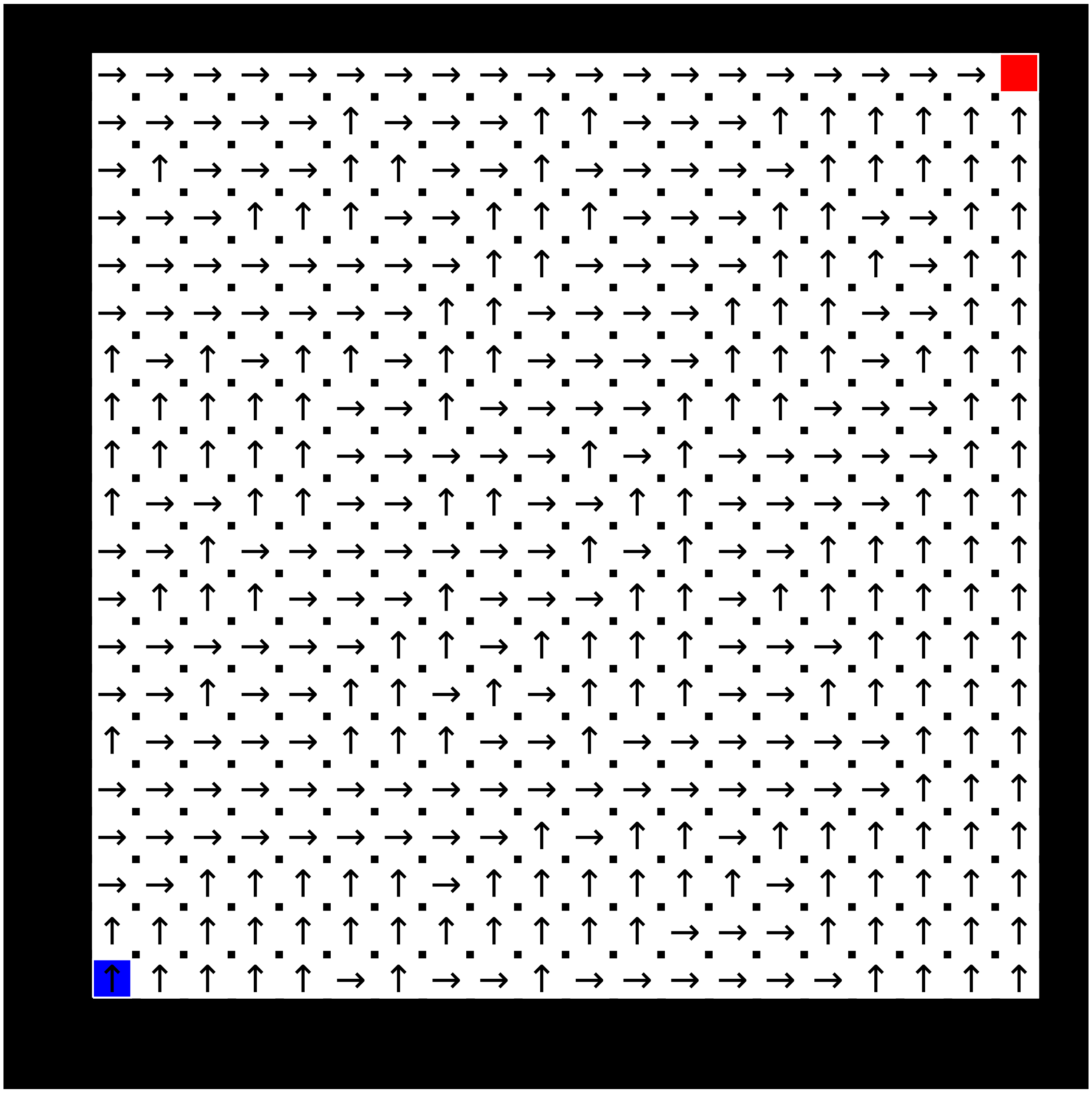


Figure 12: The solution to a 20 × 20 terrain maze, using cylic iteration and $\gamma = 0.99$. The heights for the terrain maze are shown in Fig. 13.

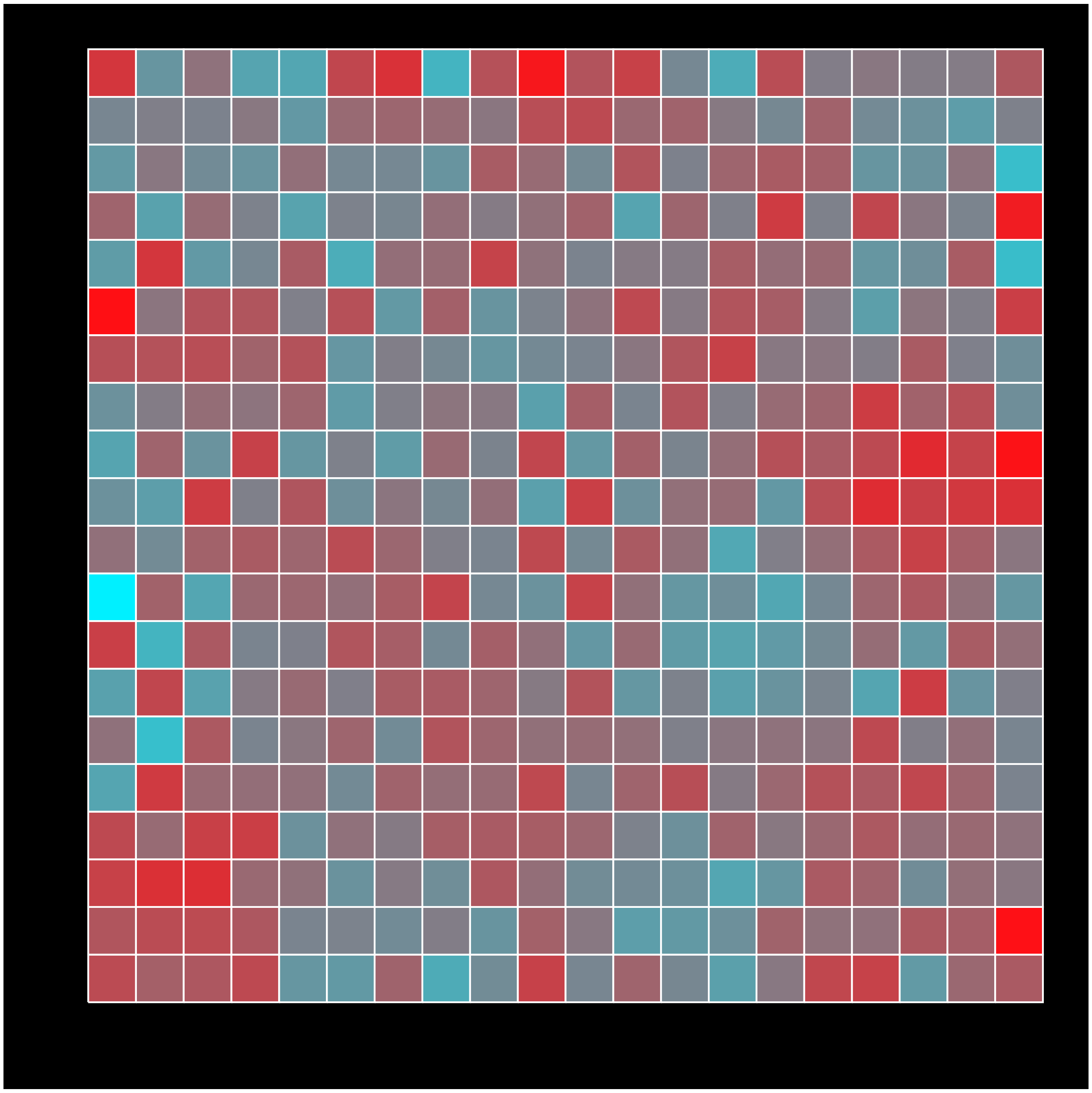


Figure 13: The heights of the terrain maze solved in Fig. 12. Redder colors are higher, and bluer colors are lower.

B Code

B.1 Utilities

```julia
using LinearAlgebra
using SparseArrays
using Random
using Plots

# NOTE: Due to limitations in how sparse matrices can be stored, even though
# positions in the maze are best thought of as a pair of integers (i, j), these
# must be converted to a single integer k. Because julia does not allow methods
# (only member functions) there is no way to do this conversion "under the
# hood." Therefore, the functions `i2p` (index to position) and `p2i` (position
# to index) are used throughout to convert between integer and index
# representations.

#=  @brief  Converts a pair of integers (i.e., a position) to an integer index
#           (i.e., an index in a matrix.)
#   @param[in]  n The dimension of the matrix into which one is indexing.
#   @param[in]  p The pair of integers representing a position in the maze.
#   @return       A single integer encoding the position.
=#
function p2i(n, p)
  n * (p[1] - 1) + p[2]
end

#=  @brief  Converts a single integer (i.e., a row or column index in a matrix)
#           to a pair of integers (i.e., a position in a maze).
#   @param[in]  n The dimension of the matrix into which one is indexing.
#   @param[in]  p The pair of integers representing a position in the maze.
#   @return       A pair of integers encoding the index.
=#
function i2p(n, i)
  if (i % n == 0)
    return (i ÷ n, n)
  else
    return (i ÷ n + 1, i % n)
  end
end

#=  @brief  Convenience object for graphs. Nodes are stored implicitly as the
#           keys of the dictionary of edges.
#
#   NOTE: All graphs are planar and rectangular, so they are also stored width a
#   height and a width.
=#
struct Graph
  edges::Dict{Tuple{Int,Int},Array{Tuple{Int,Int},1}}
  width::Int
end

#=  @brief  Convenience object for mazes. Transition probability matrices (i.e.,
# ⊠          [_j⊠^] for j ⊠⊠ _i) for location (j, k) are stored at the i-th row
#           of this.tm[j, k]. Likewise, the immediate cost for the i-th action
#           is stored as this.ic[j, k][i].
=#
struct Maze{T<:Number}
  tm::Array{SparseMatrixCSC{T,Int64},2}   # Transition matrices
  ic::Array{Array{T,1},2}                 # Action costs
  graph::Graph                            # Underlying graph
end

#=  @brief Generates a random DFS tree for a square grid.
#   @param[in]  n The width of grid.
#   @return     A graph object encoding the generated maze.
=#
function gen_random_dfs(n::Int)
  # Generate the initial set of vertices.
  nodes = [(i, j) for i = 1:n, j = 1:n]

  # Convenience functions for checking if a position in inbounds to define
  # initial edge set and generating set of neighbors.
  inbounds = (i, j) -> ((0 < i && i <= n) && (0 < j && j <= n))
  neighbors = (i, j) -> filter(
      (p) -> inbounds(p[1], p[2]),
      [(i - 1, j), (i + 1, j), (i, j - 1), (i, j + 1)]
  )
  edges = Dict([(node, neighbors(node...)) for node in nodes])
  visited = Dict([(node, false) for node in nodes])

  # Implement DFS.
  cur_node = (1, 1)
  stack = []
  dfs_tree = Dict([(node, []) for node in nodes])
  while true
    # Mark that the current node has been visited.
    visited[cur_node] = true
    # Add this node's edges to stack
    for new_node = shuffle(edges[cur_node])
      push!(stack, (cur_node, new_node))
    end

    # Get the next node by popping edges off the stack until an edge to an
    # unvisited node is found.
    while (length(stack) > 0)
      edge = pop!(stack)
      cur_node = edge[1]
      # Break the loop if e connects to an unvisited node, and add that edge to
      # the DFS tree.
      if (! visited[edge[2]])
        dfs_tree[cur_node] = vcat(dfs_tree[cur_node], edge[2])
        dfs_tree[edge[2]] = vcat(dfs_tree[edge[2]], cur_node)
        cur_node = edge[2]
        break
      end
    end
    length(stack) > 0 || break
  end

  # The final node should be a self-loop.
  dfs_tree[(n, n)] = [(n, n)]

  Graph(dfs_tree, n)
end

#=  @brief Generates a square grid graph.
#   @param[in]  n The width of grid.
#   @return     A graph object encoding the generated maze.
=#
function gen_grid(n)
  # Generate the initial set of vertices.
  nodes = [(i, j) for i = 1:n, j = 1:n]

  # Convenience functions for checking if a position in inbounds to define
  # initial edge set and generating set of neighbors.
  inbounds = (i, j) -> ((0 < i && i <= n) && (0 < j && j <= n))
  neighbors = (i, j) -> filter(
      (p) -> inbounds(p[1], p[2]),
      [(i - 1, j), (i + 1, j), (i, j - 1), (i, j + 1)]
  )
  edges = Dict([(node, neighbors(node...)) for node in nodes])

  # The final node should be a self-loop.
  edges[(n, n)] = [(n, n)]

  return Graph(edges, n)
end

#=  @brief Generates a standard n×n maze using randomized depth-first search.
#          The maze is returned as a Maze object.
#   @param[in]  g The graph of the maze.
#   @return       A maze object encoding the generated maze.
=#
function gen_standard_maze(g::Graph)
  n = g.width

  # Use the DFS tree to create a Maze object
  transition_matrices = Array{SparseMatrixCSC{Int64,Int64},2}(undef, n, n)
  for i = 1:n, j = 1:n
    # Connected edges are converted to vectors in the following way: the
    # "direction" of movement becomes the first index (encoded as an integer
    # with m2i) and the destination (encoded as an integer with p2i) becomes the
    # second index.
    incident_edges = g.edges[(i, j)]
    J = broadcast(p2i, n, incident_edges)
    I = 1:length(incident_edges)
    m = length(incident_edges)
    transition_matrices[i, j] = sparse(I, J, ones(m), m, n * n)
  end

  action_costs = Array{Array{Int64,1},2}(undef, n, n)
```

```julia
  for i = 1:n, j = 1:n
    # The only cost is -1 for arriving at the end of the maze and 1 otherwise.
    reached_end = (x) -> 2 * (x[1] == n && x[2] == n) - 1
    action_costs[i, j] = - reached_end.(g.edges[(i, j)])
  end

  # The cost for taking the self-loop at the final node should be 0.
  action_costs[n, n] = [0]

  return Maze(transition_matrices, action_costs, g)
end

#=  @brief Generates a standard n×n maze using randomized depth-first search.
#          The maze is returned as a Maze object.
#   @param[in]  n The side-length of the grid for the maze.
#   @return       A maze object encoding the generated maze.
=#
function gen_standard_maze(n::Int)
  # Generate a maze of the requested size
  g = gen_random_dfs(n)
  # Return the maze object generated from the graph
  return gen_standard_maze(g)
end

#=  @brief  Generates an n×n terrain maze, by randomly drawing a height for each
#           position. The maze is returned as a Maze object.
#   @param[in]  n             The side-length of the grid for the maze.
#   @param      return_height Whether to return the n×n matrix of heights.
#   @return     A maze object encoding the generated maze.
=#
function gen_terrain_maze(n::Int, return_height = false)
  # Generate the grid graph of the requested size
  g = gen_grid(n)

  # Each point's height is drawn from rand(0, 10) and then averaged with its
  # neighbors
  Height = 10 * randn(n, n)
  Avg_Height = zeros(n, n)
  for i = 2:(n - 1), j = 2:(n - 1)
    Avg_Height[i, j] = (Height[i + 1, j] +
                         Height[i, j + 1] +
                         Height[i - 1, j] +
                         Height[i, j - 1]) / 4
  end
  Avg_Height[1, 1:n] = (Height[1, 1:n] + Height[2, 1:n]) / 2
  Avg_Height[1:n, 1] = (Height[1:n, 1] + Height[1:n, 2]) / 2
  Avg_Height[n, 1:n] = (Height[n, 1:n] + Height[n - 1, 1:n]) / 2
  Avg_Height[1:n, n] = (Height[1:n, n] + Height[1:n, n - 1]) / 2

  # At each point, you can move to any of your neighbors, and the cost of moving
  # is equal to the slope, unless moving downhill, in which case, movement is
  # free.
  transition_matrices = Array{SparseMatrixCSC{Float64,Int64},2}(undef, n, n)
  for i = 1:n, j = 1:n
```

```
    incident_edges = g.edges[(i, j)]
    J = broadcast(p2i, n, incident_edges)
    I = 1:length(incident_edges)
    m = length(incident_edges)
    transition_matrices[i, j] = sparse(I, J, ones(m), m, n * n)
  end

  action_costs = Array{Array{Float64,1},2}(undef, n, n)
  for i = 1:n, j = 1:n
    function slope(p)
      if (p[1] == n && p[2] == n)
        return -1
      elseΔ
         = (Avg_Height[p[1], p[2]]
             - Avg_Height[i, j]
             + maximum(Avg_Height)
             - minimum(Avg_Height))
        return Δ
      end
    end
    action_costs[i, j] = slope.(g.edges[(i, j)])
  end

  # The cost for taking the self-loop at the final node should be 0.
  action_costs[n, n] = [0]

  if (return_height)
    return Maze(transition_matrices, action_costs, g), Avg_Height
  else
    return Maze(transition_matrices, action_costs, g)
  end
end

#=  @brief  Generates a visual representation of the maze encoded by the graph.
#   @param  g The DFS tree used to generate the maze.
#   @return A Plots Plot object.
=#
function plot_maze(g::Graph)
  # Get the nodes and set the plot background color to be black.
  n = g.width
  plt = plot(
    bg = :black,
    xlim = (0, n + 1),
    ylim = (0, n + 1),
    framestyle = :none,
    size = (600, 600),
    legend = false
  )

  # Add a white rectangle for each edge in the graph spanning the positions of
  # the two nodes making it up.
  nodes = keys(g.edges)
  for node = nodes
    for edge = g.edges[node]
```

```
      min_x = min(node[1], edge[1])
      max_x = max(node[1], edge[1])
      min_y = min(node[2], edge[2])
      max_y = max(node[2], edge[2])

      box = Shape([(min_x - 0.4, min_y - 0.4),
                   (min_x - 0.4, max_y + 0.4),
                   (max_x + 0.4, max_y + 0.4),
                   (max_x + 0.4, min_y - 0.4)])
      plot!(plt, box, fillcolor = :white)
    end
  end

  # Add a blue square to mark the start, and a red square to mark the finish.
  plot!(plt, Shape([(0.6, 0.6),
                    (0.6, 1.4),
                    (1.4, 1.4),
                    (1.4, 0.6)]),
        fillcolor = :blue)
  plot!(plt, Shape([(n - 0.4, n - 0.4),
                    (n - 0.4, n + 0.4),
                    (n + 0.4, n + 0.4),
                    (n + 0.4, n - 0.4)]),
        fillcolor = :red)

  return plt
end

#=  @brief  Generates a visual representation of the maze encoded by the graph.
#   @param  m A maze object.
#   @return A Plots Plot object.
=#
function plot_maze(m::Maze)
  return plot_maze(m.graph)
end

#=  @brief  Generates a visual representation of the cost-to-go at each point in
#           the maze encoded by the graph.
#   @param  m The DFS tree used to generate the maze or the maze itself.
#   @param ⊠  An n×n matrix giving the cost-to-go at each position.
=#
function plot_cost_to_go(m, ⊠)
  plt = plot_maze(m)
  n = size⊠(, 1)⊠

  _max = maximum⊠()⊠
  _min = minimum⊠()
  # Local function for mapping values to the correct heatmap color.
  function heatmap(x)
    v = (x - ⊠_min) / ⊠(_max - ⊠_min)
    RGBA(1 - v, v * 0.94 + (1 - v) * 0.06, v + (1 - v) * 0.08)
  end

  # Add a box with a heatmapped color for each position.
  for i = 1:n, j = 1:n
    box = Shape([(i - 0.2, j - 0.2),
                 (i - 0.2, j + 0.2),
                 (i + 0.2, j + 0.2),
                 (i + 0.2, j - 0.2)])
    plot!(plt, box, fillcolor = heatmap⊠([i, j]))
  end
  return plt
end

#=  @brief  Generates a visual representation of the solution to a maze encoded
#           by a cost-to-go vector (i.e., n×n matrix).
#   @param  m A maze object.
#   @param ⊠  The solution to the maze, encoded as an n×n matrix where the (i,
#           j) entry is the cost to go at position (i, j).
#   @return A Plots Plot object.
=#
function plot_solution(maze, ⊠, γ)
  soln = extract_policy(maze, ⊠, γ)
  plt = plot_maze(maze)
  n = size⊠(, 1)

  # Add an arrow pointing in the indicated direction for each position.
  for i = 1:n, j = 1:n
    arrow = ""
    if (soln[i, j] == (1, 0))
      arrow = "→"
    elseif (soln[i, j] == (-1, 0))
      arrow = "←"
    elseif (soln[i, j] == (0, 1))
      arrow = "↑"
    elseif (soln[i, j] == (0, -1))
      arrow = "↓"
    end
    annotate!(plt, i, j, arrow, :black)
  end

  return plt
end

#=  @brief  Extracts an optimal policy from a cost vector.
#   @param  maze  The maze object.
#   @param ⊠      Cost-to-go. (An n×n matrix.)
#   @param γ      Discount factor.
#   @return An n×n matrix where the (i, j) entry is the direction of movement
#           recommended by the policy.
=#
function extract_policy(maze, ⊠, γ)
  n = size⊠(, 1)
  ret = Array{Tuple{Int, Int},2}(undef, n, n)
  y = Array{Float64, 1}(undef, n * n)
  for i = 1:n, j = 1:n
    y[p2i(n, (i, j))] = ⊠[i, j]
  end
```

```julia
  for i = 1:n, j = 1:n
    a = argmin(maze.ic[i, j] + γ * maze.tm[i, j] * y)
    p = i2p(n, argmax(m.tm[i, j][a, :]))
    ret[i, j] = (p .- (i, j))
  end
  return ret
end

#= @brief Plots a subset of positions in a maze.
#  @param[in] maze    The maze object.
#  @param[in] subset  The subset of the maze, encoded as an array of states.
#  @return    A plot object visualizing which positions are in the subset.
=#
function plot_subset(maze, subset)
  plt = plot_maze(maze)

  for position in subset
    i = position[1]
    j = position[2]
    box = Shape([(i - 0.2, j - 0.2),
                 (i - 0.2, j + 0.2),
                 (i + 0.2, j + 0.2),
                 (i + 0.2, j - 0.2)])
    plot!(plt, box, fillcolor = :green)
  end
  return plt
end

function plot_terrain(A)
  n = size(A, 1)

  plt = plot(
    bg = :black,
    xlim = (0, n + 1),
    ylim = (0, n + 1),
    framestyle = :none,
    size = (600, 600),
    legend = false
  )

  function heatmap(x)
    v = (x - minimum(A)) / (maximum(A) - minimum(A))
    RGBA(1 - v, v * 0.94 + (1 - v) * 0.06, v + (1 - v) * 0.08)
  end

  # Add a box with a heatmapped color for each position.
  for i = 1:n, j = 1:n
    box = Shape([(i - 0.5, j - 0.5),
                 (i - 0.5, j + 0.5),
                 (i + 0.5, j + 0.5),
                 (i + 0.5, j - 0.5)])
    plot!(plt, box, fillcolor = heatmap(A[i, j]))
  end
  return plt
```

```
end
```

B.2 Value Iteration

```
using StatsBase

#=  @brief  Performs value iteration on the initial guess for the specificed
#           number of iterations.
#   @param[in]  maze    Maze object.
#   @param[in,out] ⊠    The inital guess of the state values. (An n×n matrix.)
#   @oaram[in] γ        The discount factor.
#   @param[in]  m       The maximum number of iterations.
#   @param[in] ⊠ _star  The true cost-to-go vector.
#   @return An n×n matrix containing the values after m iterations
#           of value iteration, along with the error relative to the true value
#           at each iteration, if provided.
=#
function vi(maze, ⊠, γ, m, ⊠_star = nothing)
  # Initialize the required variables.
  n = size⊠(, 1)
  y_old_vec = Array{Float64, 1}(undef, n * n)
  y_new_vec = Array{Float64, 1}(undef, n * n)
  for i = 1:n, j = 1:n
    y_old_vec[p2i(n, (i, j))] = ⊠[i, j]
  end
  iter = 1    # Current iteration number

  # If provided the true value, reshape it.
  if ⊠(_star != nothing)
    y_star_vec = Array{Float64, 1}(undef, n * n)
    convergence = Array{Float64, 1}(undef, m)
    for i = 1:n, j = 1:n
      y_star_vec[p2i(n, (i, j))] = ⊠_star[i, j]
    end
  end

  while true
    # Loop over the values of the MDP. At each position, update the values
    # according to the following update scheme:
    # ⊠    ^{k+1}_i = \min_{j \in ⊠_i} c_j + ⊠_j⊠^ ⊠^k
    # Stop once the desired number of iterations is reached.
    (iter <= m) || break

    for i = 1:n, j = 1:n
      k = p2i(n, (i, j))
      y_new_vec[k] = min((maze.ic[i, j] + γ * maze.tm[i, j] * y_old_vec)...)
    end

    # If the true cost-to-go vector is provided, check convergence.
    if ⊠(_star != nothing)
      convergence[iter] = norm(y_new_vec .- y_star_vec)
    end

    # Switch the old and new vectors to update y_new_vec.
    tmp = y_old_vec
    y_old_vec = y_new_vec
    y_new_vec = tmp

    # Advance to next round of iteration.
    iter += 1
  end

  # Reshape value to n×n grid.
  for i = 1:length(y_new_vec)
    p = i2p(n, i)⊠
    [p[1], p[2]] = y_new_vec[i]
  end

  # Return.
  if ⊠(_star != nothing)
    return ⊠, convergence
  else
    return ⊠
  end
end

#=  @brief  Performs randomized value iteration on the initial guess for the
#           specificed number of iterations.
#   @param[in]  maze    Maze object.
#   @param[in,out] ⊠    The inital guess of the state values. (An n×n matrix.)
#   @oaram[in] γ        The discount factor.
#   @param[in]  m       The maximum number of iterations.
#   @param[in[  p       The probability of inclusion in the subset. (Each state
#                       is included independently with probability p.)
#   @param[in] ⊠ _star  The true cost-to-go vector.
#   @return An n×n matrix containing the values after m iterations
#           of value iteration, along with the error relative to the true value
#           at each iteration, if provided.
=#
function rand_vi(maze, ⊠, γ, m, p, ⊠_star)
  # Initialize the required variables.
  n = size⊠(, 1)
  y_old_vec = Array{Float64, 1}(undef, n * n)
  for i = 1:n, j = 1:n
    y_old_vec[p2i(n, (i, j))] = ⊠[i, j]
  end
  y_new_vec = copy(y_old_vec)
  iter = 1                                          # Current iteration number
  state_vector = [(i, j) for i = 1:n for j = 1:n]   # Vector of all possible states

  # If provided the true value, reshape it.
  if ⊠(_star != nothing)
    y_star_vec = Array{Float64, 1}(undef, n * n)
    convergence = Array{Float64, 1}(undef, m)
    for i = 1:n, j = 1:n
      y_star_vec[p2i(n, (i, j))] = ⊠_star[i, j]
    end
  end

  while (true)
```

```
    # Loop over the values of the MDP. At each position, update the values
    # for randomly chosen states according to the following update scheme:
    # ⊠    ^{k+1}_i = \min_{j \in ⊠_i} c_j + ⊠_j⊠^ ⊠^k
    # Stop once the desired number of iterations is reached.
    (iter <= m) || break

    # Generate a random sample from the state vector.
    updates = randsubseq(state_vector, p)

    # Update ⊠ for every state in the subsequence.
    for state in updates
      i = state[1]
      j = state[2]
      k = p2i(n, (i, j))
      y_new_vec[k] = min((maze.ic[i, j] + γ * maze.tm[i, j] * y_old_vec)...)
    end

    # If the true cost-to-go vector is provided, check convergence.
    if ⊠(_star != nothing)
      convergence[iter] = norm(y_new_vec .- y_star_vec)
    end

    # Update y_old_vec at the chosen states for the next round of iteraion.
    # NOTE: Since not every state is updated, we can't simply exchange the
    # vectors in this implementation, and instead must copy the updates over
    # manually.
    for state in updates
      i = state[1]
      j = state[2]
      k = p2i(n, (i, j))
      y_old_vec[k] = y_new_vec[k]
    end

    # Advance to the next round of iteration.
    iter += 1
  end

  # Reshape value to n×n grid.
  for i = 1:length(y_new_vec)
    p = i2p(n, i)⊠
    [p[1], p[2]] = y_new_vec[i]
  end

  # Return.
  if ⊠(_star != nothing)
    return ⊠, convergence
  else
    return ⊠
  end
end

#=  @brief  Performs randomized, influence-tree based value iteration on the
#           initial guess for the specificed number of iterations.
#   @param[in]  maze    Maze object.
```

```julia
#   @param[in,out] ⊠    The inital guess of the state values. (An n×n matrix.)
#   @oaram[in] γ        The discount factor.
#   @param[in]  m       The maximum number of iterations.
#   @param[in[  p       The proportion of the total number of states to include
#                       at each update step.
#   @param[in] ⊠ _star  The true cost-to-go vector.
#   @return An n×n matrix containing the values after m iterations
#           of value iteration, along with the error relative to the true value
#           at each iteration, if provided.
#
# NOTE: It is necessary to generate a set of updates of a certain fixed size,
# rather than choosing elements of the subset independently with probability p,
# in case the Markov chain defined by a policy is not recurrent, in which case
# an unlucky random walk can have very undesirable sampling properties. (In
# particular, the number of states updated at each step may become 0 in finite
# time.)
=#
function infl_vi(maze, ⊠, γ, m, p, ⊠_star = nothing)
  # Initialize the required variables.
  n = size⊠(, 1)
  s = ceil(Int64, p * n^2)
  y_old_vec = Array{Float64, 1}(undef, n * n)
  for i = 1:n, j = 1:n
    y_old_vec[p2i(n, (i, j))] = ⊠[i, j]
  end
  y_new_vec = copy(y_old_vec)
  iter = 1                                          # Current iteration number
  state_vector = [(i, j) for i = 1:n for j = 1:n]   # Vector of all possible states

  # If provided the true value, reshape it.
  if ⊠(_star != nothing)
    y_star_vec = Array{Float64, 1}(undef, n * n)
    convergence = Array{Float64, 1}(undef, m)
    for i = 1:n, j = 1:n
      y_star_vec[p2i(n, (i, j))] = ⊠_star[i, j]
    end
  end

  # Closure which, given subset of states, randomly chooses a set of states
  # reachable from the current state to be updated on the next iteration.
  function infl(subset)
    infl_states = Set(vcat([maze.graph.edges[node] for node in subset]...))
    return sample(collect(infl_states), s)
  end
  updates = infl(state_vector)

  while (true)
    # Loop over the values of the MDP. At each position, update the values
    # for a randomly chosen subset of states reachable from the previous set of
    # states updated according to the following update scheme:
    # ⊠    ^{k+1}_i = \min_{j \in ⊠_i} c_j + ⊠_j⊠^ ⊠^k
    # Stop once the desired number of iterations is reached.
    (iter <= m) || break
    # Get the next set of states.
    # NOTE: To deal with non-recurrence, every 20 * s iterations, randomly reset
    # the set of updates.
    if ((iter % (20 * s)) == 0)
      updates = infl(state_vector)
    else
      updates = infl(updates)
    end

    # Update ⊠ for every state in the subsequence.
    for state in updates
      i = state[1]
      j = state[2]
      k = p2i(n, (i, j))
      y_new_vec[k] = min((maze.ic[i, j] + γ * maze.tm[i, j] * y_old_vec)...)
    end

    # If the true cost-to-go vector is provided, check convergence.
    if ⊠(_star != nothing)
      convergence[iter] = norm(y_new_vec .- y_star_vec)
    end

    # Update y_old_vec at the chosen states for the next round of iteraion.
    # NOTE: Since not every state is updated, we can't simply exchange the
    # vectors in this implementation, and instead must copy the updates over
    # manually.
    for state in updates
      i = state[1]
      j = state[2]
      k = p2i(n, (i, j))
      y_old_vec[k] = y_new_vec[k]
    end

    # Advance to the next round of iteration.
    iter += 1
  end

  # Reshape value to n×n grid.
  for i = 1:length(y_new_vec)
    p = i2p(n, i)⊠
    [p[1], p[2]] = y_new_vec[i]
  end

  # Return.
  if ⊠(_star != nothing)
    return ⊠, convergence
  else
    return ⊠
  end
end

#=  @brief  Performs cyclic value iteration on the initial guess for the
#           specified number of iterations.
#   @param[in]  maze    Maze object.
#   @param[in,out] ⊠    The inital guess of the state values. (An n×n matrix.)
#   @oaram[in] γ        The discount factor.
#   @param[in]  m       The maximum number of iterations.
#   @param[in] ⊠ _star  The true cost-to-go vector.
#   @return An n×n matrix containing the values after m iterations
#           of value iteration, along with the error relative to the true value
#           at each iteration, if provided.
=#
function cyclic_vi(maze, ⊠, γ, m, ⊠_star = nothing)
  # Initialize the required variables.
  n = size⊠(, 1)
  y_vec = Array{Float64, 1}(undef, n * n)
  for i = 1:n, j = 1:n
    y_vec[p2i(n, (i, j))] = ⊠[i, j]
  end
  iter = 1    # Current iteration number

  # If provided the true value, reshape it.
  if ⊠(_star != nothing)
    y_star_vec = Array{Float64, 1}(undef, n * n)
    convergence = Array{Float64, 1}(undef, m)
    for i = 1:n, j = 1:n
      y_star_vec[p2i(n, (i, j))] = ⊠_star[i, j]
    end
  end

  while true
    # Loop over the values of the MDP. At each position, update the values
    # according to the following update scheme:
    # ⊠˜   ^{k+1}_i = \min_{j \in ⊠_i} c_j + ⊠_j⊠^ ⊠^̃k
    # Note that updates on ⊠˜are performed sequentially, so that updates to
    # previous states are available when updating subsequent states at each
    # round of iteration. Then, at the end of iteration, set ⊠^{k+1} = ⊠^̃{k+1}.
    # Stop once the desired number of iterations is reached.
    (iter <= m) || break

    for i = 1:n, j = 1:n
      k = p2i(n, (i, j))
      y_vec[k] = min((maze.ic[i, j] + γ * maze.tm[i, j] * y_vec)...)
    end

    # If the true cost-to-go vector is provided, check convergence.
    if ⊠(_star != nothing)
      convergence[iter] = norm(y_vec .- y_star_vec)
    end

    # Advance to next round of iteration.
    iter += 1
  end

  # Reshape value to n×n grid.
  for i = 1:length(y_vec)
    p = i2p(n, i)⊠
    [p[1], p[2]] = y_vec[i]
  end

  # Return.
  if ⊠(_star != nothing)
    return ⊠, convergence
  else
    return ⊠
  end
end

#=  @brief  Performs randomized cyclic value iteration on the initial guess for
#           the specified number of iterations.
#   @param[in]  maze    Maze object.
#   @param[in,out] ⊠    The inital guess of the state values. (An n×n matrix.)
#   @oaram[in] γ        The discount factor.
#   @param[in]  m       The maximum number of iterations.
#   @param[in] ⊠ _star  The true cost-to-go vector.
#   @return An n×n matrix containing the values after m iterations
#           of value iteration, along with the error relative to the true value
#           at each iteration, if provided.
=#
function rand_cyclic_vi(maze, ⊠, γ, m, ⊠_star = nothing)
  # Initialize the required variables.
  n = size⊠(, 1)
  y_vec = Array{Float64, 1}(undef, n * n)
  for i = 1:n, j = 1:n
    y_vec[p2i(n, (i, j))] = ⊠[i, j]
  end
  iter = 1                                           # Current iteration number
  state_vector = [(i, j) for i = 1:n for j = 1:n]   # Vector of all possible states

  # If provided the true value, reshape it.
  if ⊠(_star != nothing)
    y_star_vec = Array{Float64, 1}(undef, n * n)
    convergence = Array{Float64, 1}(undef, m)
    for i = 1:n, j = 1:n
      y_star_vec[p2i(n, (i, j))] = ⊠_star[i, j]
    end
  end

  while (true)
    # Loop over the values of the MDP. At each position, update the values
    # according to the following update scheme:
    # ⊠˜   ^{k+1}_{i(j)} = \min_{j \in ⊠_{i(j)}} c_j + ⊠_j⊠^ ⊠̂k
    # where i(j) is a random permutation of 1 to n×n. Note that updates on ⊠˜
    # are performed sequentially, so that updates to previous states are
    # available when updating subsequent states at each round of iteration.
    # Then, at the end of iteration, set ⊠^{k+1} = ⊠̂{k+1}.  Stop once the
    # desired number of iterations is reached.
    (iter <= m) || break

    for state in shuffle!(state_vector)
      i = state[1]
      j = state[2]
      k = p2i(n, (i, j))
      y_vec[k] = min((maze.ic[i, j] + γ * maze.tm[i, j] * y_vec)...)
    end

    # If the true cost-to-go vector is provided, check convergence.
    if ⊠(_star != nothing)
      convergence[iter] = norm(y_vec .- y_star_vec)
    end

    # Advance to the next round of iteration.
    iter += 1
  end

  # Reshape value to n×n grid.
  for i = 1:length(y_vec)
    p = i2p(n, i)⊠
    [p[1], p[2]] = y_vec[i]
  end

  # Return.
  if ⊠(_star != nothing)
    return ⊠, convergence
  else
    return ⊠
  end
end
```

B.3 Numerical Experiments

```
using DelimitedFiles

#= @brief Performs the following benchmark test for each (square) size in the range.
#         Checks the rate of convergence over the first 10,000 iterations, and
#         writes the result as
#             "exp/{ maze_type }_ { size }_convergence.csv"
#         Note that the size of the input will be the square of the value in the
#         range.
# @param[in]  r The range of the size of the numerical experiments. (Will
#               generate mazes of both types of sizes i^2, …, k^2 for the range
#               i:k and perform numerical experiments on both.)
# @param[in] γ  The discount factor. (Defaults to 0.9.)
# @param[in]  p The probability used for random-style iterations. (Defaults to 0.5.)
# @param[in]  m The maximum number of iterations. (Defaults to 10000.)
# @return     nothing
=#
function scale_exp(r::UnitRange{Int64}, γ::Float64 = 0.9, m::Int64 = 10000,
                   p::Float64 = 0.5)
  for i = r
    mazes = Dict(["standard" => gen_standard_maze(i), "terrain" => gen_terrain_maze(i)])
    for kv in mazes
      maze_type = kv[1]
      maze = kv[2]
      s = cyclic_vi(maze, zeros(i, i), γ, m)
      _, c_vi = vi(maze, zeros(i, i), γ, m, s)
      _, c_rand_vi = rand_vi(maze, zeros(i, i), γ, m, p, s)
      _, c_infl_vi = infl_vi(maze, zeros(i, i), γ, m, p, s)
      _, c_cyclic_vi = cyclic_vi(maze, zeros(i, i), γ, m, s)
      _, c_rand_cyclic_vi = rand_cyclic_vi(maze, zeros(i, i), γ, m, s)
      writedlm("exp/$( maze_type )_$( i^2 )_convergence.csv",
               hcat(c_vi, c_rand_vi, c_infl_vi, c_cyclic_vi, c_rand_cyclic_vi,
                    fill(i^2, m)))
    end
  end
end

#= @brief Performs the following benchmark test for each (square) size and
#         inclusion parameter p in the range. Checks the rate of convergence
#         over the first 10,000 iterations, and writes the result as
#             "exp/{ maze_type }_ { size }_{ probability}_prob.csv"
#         Note that the size of the input will be the square of the value in the
#         range.
# @param[in]  r The range of the size of the numerical experiments. (Will
#               generate mazes of both types of sizes i^2, …, k^2 for the range
#               i:k and perform numerical experiments on both.)
# @param[in]  p The range of probabilities to use in the numerical experiments.
# @param[in] γ  The discount factor. (Defaults to 0.9.)
# @param[in]  m The maximum number of iterations. (Defaults to 10000.)
# @return     nothing
=#
function rand_exp(r::UnitRange{Int64}, p, γ::Float64 = 0.9,
                  m::Int64 = 10000)
  for i = r
    mazes = Dict(["standard" => gen_standard_maze(i), "terrain" => gen_terrain_maze(i)])
    for kv in mazes
      maze_type = kv[1]
      maze = kv[2]
      s = cyclic_vi(maze, zeros(i, i), γ, m)
      for q = p
        _, c_rand_vi1 = rand_vi(maze, zeros(i, i), γ, m, q, s)
        _, c_rand_vi2 = rand_vi(maze, zeros(i, i), γ, m, q, s)
        _, c_rand_vi3 = rand_vi(maze, zeros(i, i), γ, m, q, s)
        _, c_rand_vi4 = rand_vi(maze, zeros(i, i), γ, m, q, s)
        _, c_rand_vi5 = rand_vi(maze, zeros(i, i), γ, m, q, s)
        _, c_infl_vi1 = infl_vi(maze, zeros(i, i), γ, m, q, s)
        _, c_infl_vi2 = infl_vi(maze, zeros(i, i), γ, m, q, s)
        _, c_infl_vi3 = infl_vi(maze, zeros(i, i), γ, m, q, s)
        _, c_infl_vi4 = infl_vi(maze, zeros(i, i), γ, m, q, s)
        _, c_infl_vi5 = infl_vi(maze, zeros(i, i), γ, m, q, s)
        writedlm("exp/$( maze_type )_$( i^2 )_$( q )_rand.csv",
                 hcat(c_rand_vi1, c_rand_vi2, c_rand_vi3, c_rand_vi4, c_rand_vi5,
                      c_infl_vi1, c_infl_vi2, c_infl_vi3, c_infl_vi4, c_infl_vi5,
                      fill(i^2, m), fill(q, m)))
      end
    end
  end
end
```

B.4 Plots

~~~
---
title: CME 305: Project III (Plots)
author: Johann Gaebler
date: March 15, 2021
---

```{r setup}
library(tidyverse)
library(fs)
```

# Convergence

We load the data from the experiment directory.

```{r load}
standard_maze <- dir_ls("exp") %>%
  str_subset(., "standard.*convergence") %>%
  map(read_tsv, col_names = FALSE, col_types = cols(.default = "d")) %>%
  map(~mutate(., rn = row_number())) %>%
  bind_rows() %>%
  set_names(
    "vi", "vi_rand", "vi_infl", "vi_cyclic", "vi_rand_cyclic", "size", "rn"
  ) %>%
  arrange(size) %>%
  mutate(size = as.integer(size), maze_type = "standard")

terrain_maze <- dir_ls("exp") %>%
  str_subset(., "terrain.*convergence") %>%
  map(read_tsv, col_names = FALSE, col_types = cols(.default = "d")) %>%
  map(~mutate(., rn = row_number())) %>%
  bind_rows() %>%
  set_names(
    "vi", "vi_rand", "vi_infl", "vi_cyclic", "vi_rand_cyclic", "size", "rn"
  ) %>%
  arrange(size) %>%
  mutate(size = as.integer(size), maze_type = "terrain")

df_convegence <- bind_rows(standard_maze, terrain_maze) %>%
  pivot_longer(
    cols = starts_with("vi"),
    names_to = "type",
    values_to = "error"
  ) %>%
  arrange(maze_type, size, type, rn)

rm(standard_maze, terrain_maze)
```

Next, we extract the length of time required to achieve convergence to
$\varepsilon = 10^{-6}$ and $\varepsilon = 10^{-12}$.
~~~

~~~
```{r convergence}
convergence <- df_convergence %>%
  group_by(size, maze_type, type) %>%
  summarize(converge = min(rn[error < 1e-6]), .groups = "drop")

convergence %>%
  mutate(
    type = case_when(
      type == "vi" ~ "Value iteration",
      type == "vi_rand" ~ "Random value iteration",
      type == "vi_infl" ~ "Influence-tree based random value iteration",
      type == "vi_cyclic" ~ "Cyclic value iteration",
      type == "vi_rand_cyclic" ~ "Random cyclic value iteration"
    )
  ) %>%
  filter(maze_type == "terrain") %>%
  ggplot(aes(x = size, y = converge, color = type)) +
  geom_line() +
  labs(
    title = expression(paste(
      "Number of iterations required to converge to within ",
      10^{-6},
      " (Terrain Maze)"
    )),
    y = "Number of iterations",
    x = "Size of input",
    color = "Type of Value Iteration"
  )

convergence %>%
  mutate(
    type = case_when(
      type == "vi" ~ "Value iteration",
      type == "vi_rand" ~ "Random value iteration",
      type == "vi_infl" ~ "Influence-tree based random value iteration",
      type == "vi_cyclic" ~ "Cyclic value iteration",
      type == "vi_rand_cyclic" ~ "Random cyclic value iteration"
    )
  ) %>%
  filter(maze_type == "standard") %>%
  ggplot(aes(x = size, y = converge, color = type)) +
  geom_line() +
  labs(
    title = expression(paste(
      "Number of iterations required to converge to within ",
      10^{-6},
      " (Standard Maze)"
    )),
    y = "Number of iterations",
    x = "Size of input",
    color = "Type of Value Iteration"
  )
```
~~~

~~~
Next we plot the rate of convergence of the different methods.

```{r rate_of_convergence}
df_convergence %>%
  mutate(
    type = case_when(
      type == "vi" ~ "Value iteration",
      type == "vi_rand" ~ "Random value iteration",
      type == "vi_infl" ~ "Influence-tree based random value iteration",
      type == "vi_cyclic" ~ "Cyclic value iteration",
      type == "vi_rand_cyclic" ~ "Random cyclic value iteration"
    )
  ) %>%
  filter(maze_type == "terrain", rn < 300) %>%
  ggplot(aes(x = rn, y = error, color = type)) +
  geom_line() +
  facet_wrap("size") +
  labs(
    title = "Convergence Rate (Terrain Maze)",
    y = "Error",
    x = "Number of iterations",
    color = "Type of Value Iteration"
  ) +
  scale_y_log10()

df_convergence %>%
  mutate(
    type = case_when(
      type == "vi" ~ "Value iteration",
      type == "vi_rand" ~ "Random value iteration",
      type == "vi_infl" ~ "Influence-tree based random value iteration",
      type == "vi_cyclic" ~ "Cyclic value iteration",
      type == "vi_rand_cyclic" ~ "Random cyclic value iteration"
    )
  ) %>%
  filter(maze_type == "standard", rn < 300) %>%
  ggplot(aes(x = rn, y = error, color = type)) +
  geom_line() +
  facet_wrap("size") +
  labs(
    title = "Convergence Rate (Standard Maze)",
    y = "Error",
    x = "Number of iterations",
    color = "Type of Value Iteration"
  ) +
  scale_y_log10()
```

# Random Methods

Again, we load the data.

```{r load_rand}
standard_rand <- dir_ls("exp") %>%
~~~

```
  str_subset(., "standard.*rand") %>%
  map(read_tsv, col_names = FALSE, col_types = cols(.default = "d")) %>%
  map(~mutate(., rn = row_number())) %>%
  bind_rows() %>%
  set_names(
    c("rand1", "rand2", "rand3", "rand4", "rand5", "infl1", "infl2", "infl3",
    "infl4", "infl5", "size", "p", "rn")
  ) %>%
  arrange(size, p) %>%
  mutate(size = as.integer(size), maze_type = "standard")

terrain_rand <- dir_ls("exp") %>%
  str_subset(., "terrain.*rand") %>%
  map(read_tsv, col_names = FALSE, col_types = cols(.default = "d")) %>%
  map(~mutate(., rn = row_number())) %>%
  bind_rows() %>%
  set_names(
    c("rand1", "rand2", "rand3", "rand4", "rand5", "infl1", "infl2", "infl3",
    "infl4", "infl5", "size", "p", "rn")
  ) %>%
  arrange(size, p) %>%
  mutate(size = as.integer(size), maze_type = "terrain")

df_rand <- bind_rows(standard_rand, terrain_rand) %>%
  pivot_longer(
    cols = c(starts_with("rand"), starts_with("infl")),
    names_to = c("type", "sample"),
    names_pattern = "(....)(.)",
    values_to = "error"
  ) %>%
  arrange(maze_type, size, p, type, sample, rn)

rm(standard_rand, terrain_rand)
```

```{r rand_convergence}
rand_convergence <- df_rand %>%
  group_by(size, maze_type, p, type, rn) %>%
  summarize(error = mean(error), .groups = "drop_last") %>%
  summarize(converge = min(rn[error < 1e-6]), .groups = "drop")

rand_convergence %>%
  mutate(
    type = case_when(
      type == "rand" ~ "Random value iteration",
      type == "infl" ~ "Influence-tree based random value iteration"
    )
  ) %>%
  filter(maze_type == "terrain") %>%
  ggplot(aes(x = size, y = converge)) +
  geom_line(aes(color = p, group = p)) +
  labs(
    title = expression(paste(
      "Number of iterations required for average to converge to within ",
      10^{-6},
      " (Terrain Maze)"
    )),
    y = "Number of iterations",
    x = "Size of input",
    color = "Type of Value Iteration"
  ) +
  facet_wrap(~type)

rand_convergence %>%
  mutate(
    type = case_when(
      type == "rand" ~ "Random value iteration",
      type == "infl" ~ "Influence-tree based random value iteration"
    )
  ) %>%
  filter(maze_type == "standard") %>%
  ggplot(aes(x = size, y = converge)) +
  geom_line(aes(color = p, group = p)) +
  labs(
    title = expression(paste(
      "Number of iterations required for average to converge to within ",
      10^{-6},
      " (Standard Maze)"
    )),
    y = "Number of iterations",
    x = "Size of input",
    color = "Type of Value Iteration"
  ) +
  facet_wrap(~type)
```

```{r rand_convergence_rate}
df_rand %>%
  mutate(
    type = case_when(
      type == "rand" ~ "Random value iteration",
      type == "infl" ~ "Influence-tree based random value iteration"
    )
  ) %>%
  filter(maze_type == "terrain", p %in% c(0.1, 0.5, 0.95), rn < 300, type ==
         "Random value iteration") %>%
  ggplot(aes(x = rn, y = error, color = sample)) +
  geom_line(show.legend = FALSE) +
  facet_grid(rows = vars(p), cols = vars(size)) +
  labs(
    title = "Convergence Rate (Terrain Maze, Random)",
    y = "Error",
    x = "Number of iterations"
  ) +
  theme(axis.text.x = element_text(angle = 90)) +
  scale_y_log10()

df_rand %>%
  mutate(
    type = case_when(
      type == "rand" ~ "Random value iteration",
      type == "infl" ~ "Influence-tree based random value iteration"
    )
  ) %>%
  filter(maze_type == "terrain", p %in% c(0.1, 0.5, 0.95), rn < 300, type ==
         "Influence-tree based random value iteration") %>%
  ggplot(aes(x = rn, y = error, color = sample)) +
  geom_line(show.legend = FALSE) +
  facet_grid(rows = vars(p), cols = vars(size)) +
  labs(
    title = "Convergence Rate (Terrain Maze, Influence Tree)",
    y = "Error",
    x = "Number of iterations"
  ) +
  theme(axis.text.x = element_text(angle = 90)) +
  scale_y_log10()

df_rand %>%
  mutate(
    type = case_when(
      type == "rand" ~ "Random value iteration",
      type == "infl" ~ "Influence-tree based random value iteration"
    )
  ) %>%
  filter(maze_type == "standard", p %in% c(0.1, 0.5, 0.95), rn < 1000, type ==
         "Random value iteration") %>%
  ggplot(aes(x = rn, y = error, color = sample)) +
  geom_line(show.legend = FALSE) +
  facet_grid(rows = vars(p), cols = vars(size)) +
  labs(
    title = "Convergence Rate (Standard Maze, Random)",
    y = "Error",
    x = "Number of iterations"
  ) +
  theme(axis.text.x = element_text(angle = 90)) +
  scale_y_log10()

df_rand %>%
  mutate(
    type = case_when(
      type == "rand" ~ "Random value iteration",
      type == "infl" ~ "Influence-tree based random value iteration"
    )
  ) %>%
  filter(maze_type == "standard", p %in% c(0.1, 0.5, 0.95), rn < 1000, type ==
         "Influence-tree based random value iteration") %>%
  ggplot(aes(x = rn, y = error, color = sample)) +
  geom_line(show.legend = FALSE) +
  facet_grid(rows = vars(p), cols = vars(size)) +
  labs(
    title = "Convergence Rate (Standard Maze, Influence Tree)",
    y = "Error",
    x = "Number of iterations"
  ) +
  theme(axis.text.x = element_text(angle = 90)) +
  scale_y_log10()
```
```

Finally, we examine how many extra iterations are required as a function of the
probability of inclusion, p.

```
```{r extra_iterations}
rand_convergence %>%
  filter(maze_type == "terrain") %>%
  ggplot(aes(x = size, y = p * converge, color = as_factor(p))) +
  geom_jitter() +
  facet_wrap(~type) +
  labs(
    title = "Normalized number of steps to convergence (Terrain)",
    color = "Probability of inclusion (p)",
    x = "Size",
    y = "Normalized number of steps to convergence"
  )

rand_convergence %>%
  filter(maze_type == "standard") %>%
  ggplot(aes(x = size, y = p * converge, color = as_factor(p))) +
  geom_jitter() +
  facet_wrap(~type) +
  labs(
    title = "Normalized number of steps to convergence (Standard)",
    color = "Probability of inclusion (p)",
    x = "Size",
    y = "Normalized number of steps to convergence"
  )
```
```
```