```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <errno.h>
#include <math.h>

#include "utilities.h"
#include "mdp.h"

/*  Procedure
 *    value_iteration
 *
 *  Purpose
 *    Estimate utilities with iterative updates
 *
 *  Parameters
 *   p_mdp
 *   epsilon
 *   gamma
 *   utilities
 *
 *  Produces,
 *   [Nothing.]
 *
 *  Preconditions
 *    p_mdp is a pointer to a valid, complete mdp
 *    utilities points to a valid array of length p_mdp->numStates
 *    epsilon > 0
 *    0 < gamma < 1
 *
 *  Postconditions
 *    utilities[s] contains the estimated utility value for the given state
 *
 *  Authors
 *    Daniel Nanetti-Palacios
 *    Tyler Dewey
 *
 * Documentation adapted from Jerod Weinman's policy_iteration.c
 */
void value_iteration( const mdp* p_mdp, double epsilon, double gamma,
          double *utilities)
{
  // Run value iteration!

  double *updated_utilities;
  double max_utilities_change, utilities_change;
  unsigned int state, num_states;
  size_t utilities_size;

  num_states = p_mdp->numStates;
  utilities_size = sizeof(double) * num_states;

  updated_utilities = malloc(utilities_size);
  bzero(updated_utilities, utilities_size);

  do
  {
    // update the old utilities
    memcpy(utilities, updated_utilities, utilities_size);
    max_utilities_change = 0;

    for ( state = 0; state < num_states ; state++ )
    {
      double meu;
      unsigned int action;

      if (p_mdp->terminal[state]) // if this is a terminal state
      {
        // then the utility should be just the reward
        updated_utilities[state] = p_mdp->rewards[state];
      }
      else
      {
        // otherwise, it is reward + discount_rate * meu
        calc_meu(p_mdp, state, utilities, &meu, &action);

        updated_utilities[state] = p_mdp->rewards[state] + gamma * meu;
      }

      utilities_change = fabs(updated_utilities[state] - utilities[state]);

      if (utilities_change > max_utilities_change)
      {
        max_utilities_change = utilities_change;
      }
    }
  } while(!(max_utilities_change < (epsilon * (1 - gamma) / gamma)));

  // Clean up
  free(updated_utilities);
}


/*
 * Main: value_iteration gamma epsilon mdpfile
 *
 * Runs value_iteration algorithm using gamma and with max
 * error of epsilon on utilities of states using MDP in mdpfile.
 *
 * Author: Jerod Weinman
 */
int main(int argc, char* argv[])
{
  if (argc != 4)
  {
    fprintf(stderr,"Usage: %s gamma epsilon mdpfile\n",argv[0]);
    exit(EXIT_FAILURE);
  }

  // Read and process configurations
  double gamma, epsilon;
  char* endptr; // String End Location for number parsing
  mdp *p_mdp;

  // Read gamma, the discount factor, as a double
  gamma = strtod(argv[1], &endptr);

  if ( (endptr - argv[1]) < strlen(argv[1]) )
  {
    fprintf(stderr, "%s: Illegal non-numeric value in argument gamma=%s\n",
            argv[0],argv[1]);
      exit(EXIT_FAILURE);
  }

  // Read epsilon, maximum allowable state utility error, as a double
  epsilon = strtod(argv[2], &endptr);

  if ( (endptr - argv[2]) < strlen(argv[2]) )
  {
    fprintf(stderr, "%s: Illegal non-numeric value in argument epsilon=%s\n",
            argv[0],argv[2]);
      exit(EXIT_FAILURE);
  }

  // Read the MDP file (exits with message if error)
  p_mdp = mdp_read(argv[3]);
```

```
  if (NULL == p_mdp)
  { // mdp_read prints a message
    exit(EXIT_FAILURE);
  }

  // Allocate utility array
  double * utilities;

  utilities = malloc( sizeof(double) * p_mdp->numStates );

  // Verify we have memory for utility array
  if (NULL == utilities)
  {
    fprintf(stderr,
      "%s: Unable to allocate utilities (%s)",
      argv[0],
      strerror(errno));
    exit(EXIT_FAILURE);
  }

  // Run value iteration!
  value_iteration( p_mdp, epsilon, gamma, utilities );

  // Print utilities
  unsigned int state;
  for ( state=0 ; state < p_mdp->numStates ; state++)
    printf("%f\n",utilities[state]);

  // Clean up
  free (utilities);
  mdp_free(p_mdp);

  exit(EXIT_SUCCESS);
}
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <math.h>

#include "utilities.h"
#include "mdp.h"

/*  Procedure
 *    policy_evaluation
 *
 *  Purpose
 *    Iteratively estimate state utilities under a fixed policy
 *
 *  Parameters
 *   policy
 *   p_mdp
 *   epsilon
 *   gamma
 *   utilities
 *
 *  Produces,
 *   [Nothing.]
 *
 *  Preconditions
 *    policy points to a valid array of length p_mdp->numStates
 *    Each policy entry respects 0 <= policy[s] < p_mdp->numActions
 *       and policy[s] is an entry in p_mdp->actions[s]
 *    p_mdp is a pointer to a valid, complete mdp
 *    epsilon > 0
 *    0 < gamma < 1
 *    utilities points to a valid array of length p_mdp->numStates
 *
 *  Postconditions
 *    utilities[s] has been updated according to the simplified Bellman update
 *    so that no update is larger than epsilon
 *
 *  Authors
 *    Jerod Weinman (documentation & skeleton)
 *    Daniel NP & Tyler D (implementation)
 */
void policy_evaluation( const unsigned int* policy, const mdp* p_mdp,
      double epsilon, double gamma,
      double* utilities)
{
  double *updated_utilities;
  double max_utilities_change, utilities_change, eu;

  int state, num_states, utilities_size;

  num_states = p_mdp->numStates;
  utilities_size = sizeof(double) * num_states;

  updated_utilities = malloc(utilities_size);
  bzero(updated_utilities, utilities_size);

  do
  {
    max_utilities_change = 0;

    for ( state = 0 ; state < num_states ; state++ )
    {
      if (p_mdp->terminal[state]) // if this is a terminal state
      {
        // then the utility is just the reward
        updated_utilities[state] = p_mdp->rewards[state];
      }
      else
      {
        // otherwise, it's the reward plus the discounted expected utility
        // of the policy's action
        eu = calc_eu(p_mdp, state, utilities, policy[state]);

        updated_utilities[state] = p_mdp->rewards[state] + gamma * eu;
      }

      // Check if we've found a new max change in utilities
      utilities_change = fabs(updated_utilities[state] - utilities[state]);

      if (utilities_change > max_utilities_change)
      {
        max_utilities_change = utilities_change;
      }
    }

    // Update our utilities
    memcpy(utilities, updated_utilities, utilities_size);

  } while (!(max_utilities_change <= epsilon));

  // Clean up
  free(updated_utilities);
}
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <math.h>

#include "utilities.h"
#include "policy_evaluation.h"
#include "mdp.h"


/*  Procedure
 *    policy_iteration
 *
 *  Purpose
 *    Optimize policy by alternating evaluation and improvement steps
 *
 *  Parameters
 *   p_mdp
 *   epsilon
 *   gamma
 *   policy
 *
 *  Produces,
 *   [Nothing.]
 *
 *  Preconditions
 *    p_mdp is a pointer to a valid, complete mdp
 *    policy points to a valid array of length p_mdp->numStates
 *    Each policy entry respects 0 <= policy[s] < p_mdp->numActions
 *       and policy[s] is an entry in p_mdp->actions[s]
 *    epsilon > 0
 *    0 < gamma < 1
 *
 *  Postconditions
 *    policy[s] contains the optimal policy for the given mdp
 *    Each policy entry respects 0 <= policy[s] < p_mdp->numActions
 *       and policy[s] is an entry in p_mdp->actions[s]
 *
 *  Authors
 *    Jerod Weinman (documentation & skeleton)
 *    Daniel NP & Tyler D (implementation)
 */
void policy_iteration( const mdp* p_mdp, double epsilon, double gamma,
                       unsigned int *policy)
{
  double *utilities;

  double current_eu, meu;

  unsigned int state, num_states, utilities_size, unchanged,
               maximizing_action;

  num_states = p_mdp->numStates;
  utilities_size = sizeof(double) * num_states;

  utilities = malloc(utilities_size);
  bzero(utilities, utilities_size);

  do {

    unchanged = 1;

    // evaluate our current policy, storing the updated utilities
    // in utilities
    policy_evaluation(policy, p_mdp, epsilon, gamma, utilities);

    for ( state = 0; state < num_states ; state++ )
    {
      current_eu = calc_eu(p_mdp, state, utilities, policy[state]);

      calc_meu(p_mdp, state, utilities, &meu, &maximizing_action);

      if (meu > current_eu)
      {
        policy[state] = maximizing_action;
        unchanged = 0;
      }
    }

  } while (!unchanged);

  // Clean up
  free(utilities);
}

/*  Procedure
 *    randomize_policy
 *
 *  Purpose
 *    Initialize policy to random actions
 *
 *  Parameters
 *   p_mdp
 *   policy
 *
 *  Produces,
 *   [Nothing.]
 *
 *  Preconditions
 *    p_mdp is a pointer to a valid, complete mdp
 *    policy points to a valid array of length p_mdp->numStates
 *
 *  Postconditions
 *    Each policy entry respects 0 <= policy[s] < p_mdp->numActions
 *       and policy[s] is an entry in p_mdp->actions[s]
 *    when p_mdp->numAvailableActions[s] > 0.
 */
void randomize_policy( const mdp* p_mdp, unsigned int* policy)
{
  srandom(42);
  unsigned int state;
  unsigned int action;

  for ( state=0 ; state < p_mdp->numStates ; state++)
  {
    if (p_mdp->numAvailableActions[state] > 0)
    {
      action = (unsigned int)(random() % (p_mdp->numAvailableActions[state]));
      policy[state] = p_mdp->actions[state][action];
    }
  }

}

/*
 * Main: policy_iteration gamma epsilon mdpfile
 *
 * Runs policy_iteration algorithm using gamma and policy_evaluation with max
 * changes of epsilon on MDP in mdpfile.
 */
int main(int argc, char* argv[])
{
  if (argc != 4)
  {
    fprintf(stderr,"Usage: %s gamma epsilon mdpfile\n",argv[0]);
    exit(EXIT_FAILURE);
```

```c
  }

  // Read and process configurations
  double gamma, epsilon;
  char* endptr; // String End Location for number parsing
  mdp *p_mdp;

  // Read gamma, the discount factor, as a double
  gamma = strtod(argv[1], &endptr);

  if ( (endptr - argv[1])/sizeof(char) < strlen(argv[1]) )
  {
    fprintf(stderr, "%s: Illegal non-numeric value in argument gamma=%s\n",
            argv[0],argv[1]);
      exit(EXIT_FAILURE);
  }

  // Read epsilon, maximum allowable state utility error, as a double
  epsilon = strtod(argv[2], &endptr);

  if ( (endptr - argv[2])/sizeof(char) < strlen(argv[2]) )
  {
    fprintf(stderr, "%s: Illegal non-numeric value in argument epsilon=%s\n",
            argv[0],argv[2]);
      exit(EXIT_FAILURE);
  }

  // Read the MDP file (exits with message if error)
  p_mdp = mdp_read(argv[3]);

  if (NULL == p_mdp)
  { // mdp_read prints a message
    exit(EXIT_FAILURE);
  }

  // Allocate policy array
  unsigned int * policy;

  policy = malloc( sizeof(unsigned int) * p_mdp->numStates );

  if (NULL == policy)
  {
    fprintf(stderr,
            "%s: Unable to allocate policy (%s)",
            argv[0],
            strerror(errno));
    exit(EXIT_FAILURE);
  }

  // Initialize random policy
  randomize_policy(p_mdp, policy);

  // Run policy iteration!
  policy_iteration ( p_mdp, epsilon, gamma, policy);

  // Print policies
  unsigned int state;
  for ( state=0 ; state < p_mdp->numStates ; state++)
    if (p_mdp->numAvailableActions[state])
      printf("%u\n",policy[state]);
    else
      printf("0\n",policy[state]);

  // Clean up
  free (policy);
  mdp_free(p_mdp);

}
```

```c
#include <math.h>
#include "mdp.h"
#include "utilities.h"

/*  Procedure
 *    calc_eu
 *
 *  Purpose
 *    Calculate the expected utility of a state and action in an MDP
 *
 *  Parameters
 *   p_mdp
 *   state
 *   utilities
 *   action
 *
 *  Produces
 *   eu
 *
 *  Preconditions
 *    p_mdp points to a valid mdp struc
 *    0 <= state < p_mdp->numStates
 *    utilities points to a valid array of length p_mdp->numStates
 *
 *  Practica
 *    The following are not preconditions, per se, but lend themselves
 *    to meaningful results:
 *      p_mdp->terminal[state] = 0 (No meaningful actions in a terminal state)
 *      action belongs to the array p_mdp->actions[state]
 *
 *  Postconditions
 *    eu is the average utility of subsequent states that arise from taking the
 *    specified action in the given state:
 *       eu = sum_{s'=0..p_mdp->numStates} P(s'|state,action) * utilities(s')
 *    where P(s'|s,a) represents the transition probability in the MDP
 *
 *  Authors
 *    Jerod Weinman (documentation & skeleton)
 *    Daniel NP & Tyler D (implementation)
 */
double calc_eu( const mdp*  p_mdp, unsigned int state, const double* utilities,
              const unsigned int action)
{
  double eu;   // Expected utility
  int successor;

  // if a state has no successors
  if (p_mdp->terminal[state] || p_mdp->numAvailableActions[state] <= 0)
  {
    return 0; // any action has no expected utility
  }

  eu = 0;

  // Calculate expected utility: sum_{s'} P(s'|s,a)*U(s')

  // Go through every successor state
  for (successor = 0 ; successor < p_mdp->numStates ; successor++)
  {
     eu += p_mdp->transitionProb[successor][state][action] * utilities[successor];
  }

  return eu;
}

/*  Procedure
 *    calc_meu
 *
 *  Purpose
 *    Calculate the action of maximum expected utility of a state in an MDP
 *
 *  Parameters
 *   p_mdp
 *   state
 *   utilities
 *   meu
 *   action
 *
 *  Produces
 *   [Nothing.]
 *
 *  Preconditions
 *    p_mdp points to a valid mdp struc
 *    0 <= state < p_mdp->numStates
 *    utilities points to a valid array of length p_mdp->numStates
 *    meu != NULL
 *    action != NULL
 *
 *  Postconditions
 *    *meu is the maximum expected utility of state in p_mdp:
 *        max_{a} EU(state,a)
 *      = max_{a} sum_{s'=0..p_mdp->numStates} P(s'|state,a) *
 *                                                    utilities(s')
 *    where P(s'|s,a) represents the transition probability in the MDP
 *
 *    *action is a value of a that yields *meu: argmax_{a} EU(state,a)
 *
 *  Authors
 *    Jerod Weinman (documentation & skeleton)
 *    Daniel NP & Tyler D (implementation)
 */
void calc_meu( const mdp* p_mdp, unsigned int state, const double* utilities,
               double *meu, unsigned int *action )
{
  // Calculated maximum expected utility (use calc_eu):
  unsigned int i, current_action, num_available_actions, max_action;
  unsigned int *available_actions;
  double eu, max_eu;

  num_available_actions = p_mdp->numAvailableActions[state];
  available_actions = p_mdp->actions[state];

  max_eu = -INFINITY;
  max_action = 0;

  if (num_available_actions == 0) {
    *meu = 0; // max utility of no actions is zero
    *action = 0;
    return;
  }

  for (i = 0 ; i < num_available_actions ; i++)
  {
    current_action = available_actions[i];

    eu = calc_eu(p_mdp, state, utilities, current_action);

    if (eu > max_eu)
    {
      max_eu = eu;
      max_action = current_action;
    }
  }

  *meu = max_eu;
  *action = max_action;
}
```

```
Script started on Thu 10 Dec 2015 09:01:13 PM CST
flowers$ make u\033[Ktilities
gcc -g -std=gnu99 -c mdp.c
gcc -g -std=gnu99 -c utilities.c
flowers$ make value
gcc -g -std=gnu99 -c mdp.c
gcc -g -std=gnu99 -c utilities.c
gcc -g -std=gnu99 -o value_iteration value_iteration.c  mdp.o utilities.o
flowers$ ./value_iteration .99999 .001 4x3.mdp
0.811522
0.761512
0.705252
0.867784
-0.040000
0.655243
0.917795
0.660255
0.611348
1.000000
-1.000000
0.387860
flowers$ ./value_iteration .99999 .001 16x4.mdp
-0.231684
-0.040000
-0.040000
-1.000000
-0.181686
-0.040000
-0.104063
-0.248053
-0.131688
-0.080508
-0.036064
-0.040000
-0.091133
-0.036064
0.019491
0.076436
-0.040000
-0.040000
-0.040000
0.133556
0.339819
0.289815
0.239811
0.183558
0.396074
-0.040000
-0.040000
-0.040000
0.446079
-0.040000
-0.040000
-1.000000
0.496085
0.558594
0.608601
0.578041
0.446079
-0.040000
0.668680
0.624228
-0.040000
0.674237
0.724246
0.668680
-1.000000
-0.040000
0.787452
-0.040000
0.943736
0.893725
0.837462
0.798565
1.000000
-0.040000
-0.040000
0.843714
-0.040000
-0.040000
-0.040000
0.893725
-3999.999000
-0.040000
1.000000
0.943736
flowers$ exit

Script done on Thu 10 Dec 2015 09:03:16 PM CST
```

```
Script started on Thu 10 Dec 2015 09:04:05 PM CST
flowers$ make policy
gcc -g -std=gnu99 -c mdp.c
gcc -g -std=gnu99 -c utilities.c
gcc -g -std=gnu99 -c policy_evaluation.c
gcc -g -std=gnu99 -o policy_iteration policy_iteration.c  \
mdp.o utilities.o policy_evaluation.o
flowers$ ./policy_iteration .999 .001 4x3.mdp
3
0
0
3
0
2
3
0
2
0
0
2
flowers$ ./policy_iteration .999 .001 16x4.mdp
3
0
0
0
3
0
3
0
1
1
3
0
1
1
1
3
0
0
0
3
3
0
0
0
3
0
0
0
3
0
0
0
1
1
3
3
2
0
3
3
0
1
3
0
0
0
3
0
3
0
0
3
0
0
0
3
0
0
0
3
0
0
0
0
flowers$ exit

Script done on Thu 10 Dec 2015 09:05:31 PM CST
```

Lab 11 Analysis

Daniel “NP-Complete” Nanetti-Palacios, *Box 4426*
Tyler “Dew-while loop” Dewey, *Box 3426*

Problem 5: *Value Iteration*

Tests:

```
$ ./value_iteration .99 .01 4x3.mdp
0.776184
0.716627
0.650616
0.843935
-0.040000
0.592541
0.905096
0.641327
0.560027
1.000000
-1.000000
0.337952

$ ./value_iteration .99999 .001 4x3.mdp
0.811522
0.761512
0.705252
0.867784
-0.040000
0.655243
0.917795
0.660255
0.611348
1.000000
-1.000000
0.387860
```

Predict:

We expect to see a direct path to a +1 terminal state that avoids coming close to -1 terminal states to have a greater cumulative value than both the shortest path and longer paths. In all instances distance dulls the effect of the terminal state's reward. States closer to -1 terminal states have a lower value than states farther away, and similarly, states closer to +1 terminal states have a higher value. We expect the state adjacent to both a +1 and a -1 (12,0) to have a value near 0.5. In the tests we ran, the blank state had a permanent value of -0.04, and we expect the same for all the blank states in this grid.

Experiment:

With blanks:

	0	1	2	3	4	5	6	7	
0	-0.232	-0.182	-0.132	-0.091	-0.040	0.340	0.396	0.446	Wrapped
1	-0.040	-0.040	-0.081	-0.036	-0.040	0.290	-0.040	-0.040	to
2	-0.040	-0.104	-0.036	0.019	-0.040	0.240	-0.040	-0.040	next
3	-1.000	-0.248	-0.040	0.076	0.134	0.184	-0.040	-1.000	line

	8	9	10	11	12	13	14	15
0	0.496	0.446	-0.040	-1.000	0.944	1.000	-0.040	-3999.9
1	0.559	-0.040	0.674	-0.040	0.894	-0.040	-0.040	-0.040
2	0.609	0.669	0.724	0.787	0.837	-0.040	-0.040	1.000
3	0.578	0.624	0.669	-0.040	0.799	0.844	0.894	0.944

Without blanks:

	0	1	2	3	4	5	6	7	
0	-0.232	-0.182	-0.132	-0.091	[redacted]	0.340	0.396	0.446	Wrapped
1	[redacted]	[redacted]	-0.081	-0.036	[redacted]	0.290	[redacted]	[redacted]	to
2	[redacted]	-0.104	-0.036	0.019	[redacted]	0.240	[redacted]	[redacted]	next
3	-1.000	-0.248	[redacted]	0.076	0.134	0.184	[redacted]	-1.000	line

	8	9	10	11	12	13	14	15
0	0.496	0.446	[redacted]	-1.000	0.944	1.000	[redacted]	-3999.9
1	0.559	[redacted]	0.674	[redacted]	0.894	[redacted]	[redacted]	[redacted]
2	0.609	0.669	0.724	0.787	0.837	[redacted]	[redacted]	1.000
3	0.578	0.624	0.669	[redacted]	0.799	0.844	0.894	0.944

Reflect:

Our prediction was correct that states closer to -1 terminals have lower value than those farther away (reverse for +1 terminal states). We were also correct in assuming that blank squares would have a value of -0.04 (their reward). We were very wrong about the state (12, 0), which has a value the same as the the other next-to-plus-1-terminal state. This does make sense in hindsight, because there is no way you could end up in (11,0) (the -1 terminal) from (12,0) if you move towards (13,0) (the +1 terminal). We were also surprised that if you follow the path of maximum increase that you end up going to (12, 0) rather than (15, 2), with the large difference between (12, 1) = 0,894 and (12, 3) = 0.799. In hindsight, this also makes sense (especially with our previous observation) and is consistent with our rule that states farther away from a +1 terminal have a lower value. We were also surprised and amused that (15, 0) had such a negative value.

Problem 6: Policy Iteration

Test:

```
$ ./policy_iteration .99 .01 4x3.mdp
3
0
0                 → → → ↑
3                 ↑ ↑ ↑ ↑
0                 ↑ ← ↑ ←
2
3                 → → → ↑
0                 ↑ ■ ↑ ↑
0                 ↑ ← ↑ ←
0
0
2

$ ./policy_iteration .999 .001 4x3.mdp
3
0
0                  → → → ↑
3                  ↑ ↑ ↑ ↑
0                  ↑ ← ← ←
2
3
0                  → → → +
2                  ↑ ■ ↑ -
0                  ↑ ← ← ←
0
2
```

Predict:

We observed in our tests that the policy is very risk-averse. When placed in the bottom left corner, rather than just going back up, the policy for 4x3 takes an agent all the way back to (0,2) (the bottom left corner). We assume that the policy generated for 16x4 would be similar, taking great lengths to find the "safest" path to a +1, avoiding paths that could (because of the action's unpredictable result) put the agent in or near a -1 terminal state. We predict that there will be a main path that never gets closer than it must to -1 states, where the states nearer to the -1 point towards that path. We predict that the policy will favor the +1 state farther away from the -1, not the one at (13, 0) that is separated from a -1 only by 1 square.

Results:

With blank spaces:

→	→	↓	↓	↑	→	→	→	↓	←	↑	↑	→	↑	↑	↑
↑	↑	↓	↓	↑	↑	↑	↑	↓	↑	↓	↑	↑	↑	↑	↑
↑	→	→	↓	↑	↑	↑	↑	→	→	→	→	↑	↑	↑	↑
↑	↑	↑	→	→	↑	↑	↑	→	→	↑	↑	→	→	→	↑

Without blank spaces:

→	→	↓	↓	■	→	→	→	↓	←	■	↑	→	↑	■	↑
■	■	↓	↓	■	↑	■	■	↓	■	↓	■	↑	■	■	■
■	→	→	↓	■	↑	■	■	→	→	→	→	↑	■	■	↑
↑	↑	■	→	→	↑	■	↑	→	→	↑	■	→	→	→	↑

W/o blanks & with terminal states:

→	→	↓	↓	■	→	→	→	↓	←	■	−	→	+	■	↑
■	■	↓	↓	■	↑	■	■	↓	■	↓	■	↑	■	■	■
■	→	→	↓	■	↑	■	■	→	→	→	→	↑	■	■	+
−	↑	■	→	→	↑	■	−	→	→	↑	■	→	→	→	↑

Reflect:

We were correct to some degree about the agent following the safest path. In each state near -1 terminals, the policy says to move away, which we predicted. Similar to our results in Value Iteration, our predictions were also wrong when dealing with the space (12, 0) that lies between a -1 terminal and +1 terminal states. We believed the policy of that state would direct the agent away from this area to follow a safer path to the +1 terminal at (15, 2), but we were wrong since there would be no way it would end up in the negative terminal if it takes the rational action of going towards the +1 terminal.