May 3, 2009 at 20:39

1. *Jack's Car Rental.* [From example 4.2 in *Reinforcement Learning: An Introduction*, by Sutton and Barto (1998)] Jack manages two locations for a nationwide car rental company. Each day, some number of customers arrive at each location to rent cars. If Jack has a car available, he rents it out and is credited \$10 by the national company. If he is out of cars at that location, then the business is lost. Cars become available for renting the day after they are returned. To help ensure that cars are available where they are needed, Jack can move them between the two locations overnight, at a cost of \$2 per car moved. We assume that the number of cars requested and returned at each location are poisson random variables, meaning that the probability that the number is n is $e^{-\lambda}\lambda^n/n!$, where λ is the expected number. Suppose λ is 3 and 4 for rental requests at the first and second locations and 3 and 2 for dropoffs. To simplify the problem slightly, we assume that there can be no more than 20 cars at each location (any additional cars are returned to the nationwide company, and thus disappear from the problem) and a maximum of 5 cars can be moved from one location to the other in one night. We take the discount rate to be $\gamma = 0.9$ and formulate this as a continual finite MDP, where the time steps are days, the state is $s = (n_1, n_2)$, the number of cars at each location *at the end of the day*, and the actions a are the net number of cars moved between the two locations overnight, a number between -5 and $+5$. We'll start from the policy that never moves any cars.

2. Here is an overview of the file `jack.c` that is defined by this `CWEB` program `jack.w`:

```
⟨ Header files to include 5 ⟩
⟨ Constants 6 ⟩
⟨ Global variables 7 ⟩
⟨ Function Declarations 26 ⟩
⟨ Function Definitions 12 ⟩
⟨ The main program 3 ⟩
```

3. The general layout of the *main* function is

```
⟨ The main program 3 ⟩ ≡
  int main()
  {
    ⟨ Variables local to main 10 ⟩
    ⟨ Initialization 11 ⟩
    ⟨ Policy Iteration 4 ⟩
    ⟨ Print out the policy 24 ⟩;
    return 0;
  }
```

This code is used in section 2.

4. Policy iteration is a sequence of policy evaluation and policy improvement. The sequence is finished when policy improvement doesn't change the previous policy.

```
⟨ Policy Iteration 4 ⟩ ≡
  bool has_changed;    /* Flag used to track changes in policy */
  do {
    ⟨ Iterative policy evaluation 17 ⟩;
    ⟨ Improve the policy; assign true to has_changed if the policy changed, false if it did not 22 ⟩;
  } while (has_changed);
```

This code is used in section 3.

5. We must include the standard I/O definitions, since we want to send formatted output to *stdout* and *stderr*. We also should include the *math* and *algorithm* libraries since they contain some functions we'll need.

⟨ Header files to include 5 ⟩ ≡

```
#include <stdio.h>
#include <math.h>
#include <algorithm>
  using namespace std;    /* for cleaner code (min and max functions) */
```

This code is used in section 2.

6. We'll use several constants. First, *ncar_states* is the number of cars one location may have at the end of the day, including the *state* "0 cars". The cardinality of the entire state space is $ncar\_states^2$. We'll also need to define *max_moves*, the maximum number of cars Jack is allowed to move from one location to the other at night. We also define *discount*, the discounting factor γ, and *theta*, the precision limit desired in our convergence conditions.

⟨ Constants 6 ⟩ ≡

```
  const int ncar_states = 21;
  const int max_moves = 5;
  const int max_morning = ncar_states + max_moves;
  const double discount = 0.9;
  const double theta = 1 · 10^{-7};    /* stop when differences are of order theta */
```

This code is used in section 2.

7. Probabilities $\mathcal{P}^a_{ss'}$ will be calculated from 2 two-dimensional arrays, called *prob_1* and *prob_2* (one for each location). These arrays have dimension *max_morning* × *ncar_states* and their elements contain the probability of transition at one given location, from morning to evening. So, *prob_1* [*n1* , *new_n1*] would give the probability that the number of cars at location 1 at the end of the day is *new_n1* , given that it starts the day with *n1* cars. Similarly for elements of *prob_2* .

⟨ Global variables 7 ⟩ ≡

```
  double prob_1[max_morning][ncar_states];    /* 26 x 21 */
  double prob_2[max_morning][ncar_states];
```

See also sections 8 and 9.

This code is used in section 2.

8. We'll also need to hold the expected immediate rewards, $\mathcal{R}^a_{ss'}$. The following array will be useful to calculate the immediate rewards. If we define *rew_1* as a *max_morning* array, element *rew_1* [*n1*] contains the expected —immediate— reward due to satisfied requests at location 1, given that the day starts with *n1* cars at location 1. And similarly for *rew_2* .

⟨ Global variables 7 ⟩ +≡

```
  double rew_1[max_morning];
  double rew_2[max_morning];
```

9. And finally let's define the value function and the policy. Notice that in this problem the policy is deterministic, so we may define $\pi(s)$ as the *action* a taken in state s, according to the policy π. Since we may represent states as a *ncar_states* × *ncar_states* matrix, $V(s)$ as well as $\pi(s)$ are functions defined on a matrix of that size.

⟨ Global variables 7 ⟩ +≡

```
  double V[ncar_states][ncar_states];
  int policy[ncar_states][ncar_states];
```

10. We set the λ parameters of the 4 Poisson distributions: *lmbda_1r* and *lmbda_1d* are the means of rental requests and dropoffs (returns) at location 1, respectively. And similarly for *lmbda_2r* and *lmbda_2d*, for location 2.

⟨ Variables local to *main* 10 ⟩ ≡

```
double lmbda_1r = 3.0;   /* Request rate at location 1 */
double lmbda_1d = 3.0;   /* Dropoff rate at location 1 */
double lmbda_2r = 4.0;   /* Request rate at location 2 */
double lmbda_2d = 2.0;   /* Dropoff rate at location 2 */
```

This code is used in section 3.

11. We can go now to the *main* program. In C++ arrays are initialized to 0 automatically, so there is no need to set explicitly to 0.0 all the elements in *prob_1*, *prob_2*, *rew_1*, *rew_2*, *V* and *policy*. Keeping in mind that, and once all λ's are specified, we can initialize arrays *prob_1*, *prob_2*, *rew_1* and *rew_2* to the proper values with the help of function *load_probs_rewards* and the different λ parameters.

⟨ Initialization 11 ⟩ ≡

```
load_probs_rewards(prob_1, rew_1, lmbda_1r, lmbda_1d);   /* 1st location */
load_probs_rewards(prob_2, rew_2, lmbda_2r, lmbda_2d);   /* 2nd location */
```

This code is used in section 3.

12. Before going into the details of the *load_probs_rewards* function it is useful to define several basic functions. First, we'll need the factorial function of n, $n!$:

⟨ Function Definitions 12 ⟩ ≡

```
double factorial(int n)
{
  if (n > 0) return (n * factorial(n - 1));
  else return (1.0);
}
```

See also sections 13, 14, 18, 20, 21, 23, and 25.

This code is used in section 2.

13. We'll also need a function that returns probability of n events according to the Poisson distribution with parameter *lambda*, $e^{-\lambda}\lambda^n/n!$. We use the *factorial* function defined above.

⟨ Function Definitions 12 ⟩ +≡

```
double poisson(int n, double lambda)
{
  return (exp(-lambda) * pow(lambda, (double) n)/factorial(n));
}
```

14. ⟨ Function Definitions 12 ⟩ +≡

```
void load_probs_rewards(double probs[max_morning][ncar_states], double rewards[max_morning], double
        l_reqsts, double l_drpffs)
{
  double req_prob;
  double drp_prob;
  int satisfied_req;
  int new_n;
  for (int req = 0; (req_prob = poisson(req, l_reqsts)) > theta; req++) {
    ⟨ Fill the reward matrix rewards[max_morning] using the probability req_prob 15 ⟩
    for (int drp = 0; (drp_prob = poisson(drp, l_drpffs)) > theta; drp++) {
      ⟨ Fill the probability matrix probs[max_morning][ncar_states] 16 ⟩
    }
  }
}
```

15. There is an upper limit in the amount of reward received from requests. This limit is given by the number of cars available. Also, the array of reward depends only on the number of requests (dropoffs are here irrelevant).

⟨ Fill the reward matrix *rewards*[*max_morning*] using the probability *req_prob* 15 ⟩ ≡

```
for (int n = 0; n < max_morning; n++) {
  satisfied_req = min(req, n);    /* at most, all the cars available */
  rewards[n] += 10 * req_prob * satisfied_req;    /* +10 is the reward per request */
}
```

This code is used in section 14.

16. For the calculation of the probability matrix the number of requests as well as dropoffs must be considered. There are different combinations of requests and dropoffs that lead to the same final state s'. Here we sweep all the requests and dropoffs with significant probabilities and sum the joint probability to the corresponding matrix element, which represents a possible transition.

⟨ Fill the probability matrix *probs*[*max_morning*][*ncar_states*] 16 ⟩ ≡

```
for (int m = 0; m < max_morning; m++) {
  satisfied_req = min(req, m);
  new_n = m + drp - satisfied_req;
  new_n = max(new_n, 0);    /* 0 at least */
  new_n = min(20, new_n);    /* 20 at most */
  probs[m][new_n] += req_prob * drp_prob;    /* add up the joint probability */
}
```

This code is used in section 14.

17. After rewards and probabilities have been set and variables have been initialized, *policy_eval*( ) can *evaluate* the current policy.

⟨ Iterative policy evaluation 17 ⟩ ≡

```
policy_eval();
```

This code is used in section 4.

18. ⟨ Function Definitions 12 ⟩ +≡

```
void policy_eval()
{
  double val_tmp;
  double diff;
  int a;

  do {
    diff = 0.0;
    for (int n1 = 0; n1 < ncar_states; n1++) {
      for (int n2 = 0; n2 < ncar_states; n2++) {
        ⟨ Assign the new value for each state; keep in diff the highest update difference 19 ⟩;
      }
    }
  } while (diff > theta);
}
```

19. For each state s, we must update values according to the current policy. We use here a function, *backup_action*(*n1*, *n2*, a), wich calculates the expected reward for a given state s and action a, or action-value function $Q_k(s, a)$. We also store in *diff* the highest difference $\mid V_{k+1}(s) - V_k(s) \mid$ occured during the loop over all the states.

⟨ Assign the new value for each state; keep in *diff* the highest update difference 19 ⟩ ≡

```
val_tmp = V[n1][n2];
a = policy[n1][n2];
V[n1][n2] = backup_action(n1, n2, a);
diff = max(diff, fabs(V[n1][n2] - val_tmp));
```

This code is used in section 18.

20. The function *backup_action*(*n1*, *n2*, a) uses the current value function approximation V_k. We know the deterministic penalty of taking action a, which is $-2a$. The possible incomes due to taking the same action are, on the other hand, of probabilistic nature: $\sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V_k(s')]$. Notice also that $\mathcal{R}^a_{ss'}$, with $s = (n1, n2)$ and a fixed, can be expressed in terms of *rew_1* and *rew_2*, namely *rew_1*[*n1* − a] + *rew_2*[*n2* + a].

⟨ Function Definitions 12 ⟩ +≡

```
double backup_action(int n1, int n2, int a)
{
  double val;    /* Determine the range of possible actions for the given state */

  a = min(a, +n1);
  a = max(a, -n2);
  a = min(+5, a);
  a = max(-5, a);
  val = -2 * fabs((double) a);

  int morning_n1 = n1 - a;
  int morning_n2 = n2 + a;

  for (int new_n1 = 0; new_n1 < ncar_states; new_n1++) {
    for (int new_n2 = 0; new_n2 < ncar_states; new_n2++) {
      val += prob_1[morning_n1][new_n1] * prob_2[morning_n2][new_n2] * (rew_1[morning_n1] +
            rew_1[morning_n2] + discount * V[new_n1][new_n2]);
    }
  }
  return val;
}
```

21. The function *update_policy_t*( ) takes care of improving the current policy. I returns *true* if the policy has changed with respect to the previous iteration.

⟨ Function Definitions 12 ⟩ +≡

```
bool update_policy_t()
{
  int b;
  bool has_changed = false;

  for (int n1 = 0; n1 < ncar_states; n1++) {
    for (int n2 = 0; n2 < ncar_states; n2++) {
      b = policy[n1][n2];
      policy[n1][n2] = greedy_policy(n1, n2);
      if (b ≠ policy[n1][n2]) {
        has_changed = true;
      }
    }
  }
  return (has_changed);
}
```

22. In the main function policy improvement is made with *update_policy_t*( ).

⟨ Improve the policy; assign *true* to *has_changed* if the policy changed, *false* if it did not 22 ⟩ ≡

```
has_changed = update_policy_t();
```

This code is used in section 4.

23. ⟨ Function Definitions 12 ⟩ +≡

```
int greedy_policy(int n1, int n2)
{    /* Set the range of available actions, a ∈ 𝒜(s) */
  int a_min = max(−5, −n2);
  int a_max = min(+5, +n1);
  double val;
  double best_val;
  int best_action;
  int a;

  a = a_min;
  best_action = a_min;
  best_val = backup_action(n1, n2, a);
  for (a = a_min + 1; a ≤ a_max; a++) {
    val = backup_action(n1, n2, a);
    if (val > best_val + 1 · 10^−9) {
      best_val = val;
      best_action = a;
    }
  }
  return (best_action);
}
```

24. We would like to see the optimal policy.

⟨ Print out the policy 24 ⟩ ≡

```
print_policy();
```

This code is used in section 3.

25. ⟨ Function Definitions 12 ⟩ +≡

```
void print_policy()
{
  printf("\nPolicy:\n");
  for (int n1 = 0; n1 < ncar_states; n1++) {
    printf("\n");
    for (int n2 = 0; n2 < ncar_states; n2++) {
      printf("% 2d ", policy[ncar_states - (n1 + 1)][n2]);
    }
  }
  printf("\n\n");
}
```

26. All in all the function prototypes for this program are the following

⟨ Function Declarations 26 ⟩ ≡

```
double factorial(int n);
double poisson(int n, double l);
void load_probs_rewards(double probs[max_morning][ncar_states], double rewards[max_morning], double
    l_reqsts, double l_drpffs);
void policy_eval();
double backup_action(int n1, int n2, int a);
int greedy_policy(int n1, int n2);
bool update_policy_t();
```

This code is used in section 2.

27. Index. Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. Error messages are also shown.