CMSC 421, Spring 2010: Project 3

Last updated April 28, 2010. The changes include the new due date and late date, the *time-limit* argument for the `start-game` function, some examples of programming style, and the URL of the control program.

- Due date/time: Noon on May 7
- Late date/time (5-point penalty): Noon on May 9

In Project 2, you wrote a program to return complete paths for racetrack games. But in a real-life road race, there are many reasons why it isn't feasible to compute a complete path all at once. One of them is that the car is already moving, and will crash into a wall unless we can decide our next move very quickly. Thus a somewhat more realistic model would be to choose the first move quickly (e.g., within a fixed amount of time) and make that move, then choose the second move quickly and make that move, and so forth. In this project, you'll write Lisp code to do that.

Functions to Write

1. Write a function `(start-game` *boundary finish-line time-limit*`)` that returns `NIL`. The purpose of this function is to specify that a new racetrack game is starting on a racetrack with boundary *boundary* and finish line *finish-line*.
2. Write a function `(next-velocity` *state time-limit*`)` that returns the velocity you want the racecar to use next, given that its current state is *state*.

In both functions, *time-limit* is the maximum amount of CPU time available (i.e., `start-game` and `next-velocity` each should return within *time-limit* milliseconds). I don't intend to change the value of *time-limit* during a game, but I'll use different values of *time-limit* in different games.

How your functions work is up to you. The two main challenges are (1) figuring out what move you think is best, and (2) doing so quickly. You'll need to use your creativity for this! If your program works well enough, you'll get extra credit (see page 3).

One possibility would be to modify your code from Project 2 so that it periodically checks to see how much CPU time has elapsed. When the elapsed time gets close to *time-limit*, the algorithm could interrupt its search, try to figure out which of the possible next states looks best based on the search it has done so far, and return that state.

Data Formats and Programming Style

The data formats (points, edges, velocities, states, paths, etc.) will be the same as in Project 2.

As usual, you should comment your code, indent it appropriately, and use good programming style (e.g., as in Norvig's *Tutorial on Good Lisp Programming Style*). Here are a couple of examples:

Declare your variables. For Project 2, some of you never declared any of your variables. If the test results that I sent you for Project 2 contained lots of messages like the following, then you're one of those people:

```
Warning:  Free reference to undeclared variable FOO assumed special.
```

That's bad programming style, for reasons that I hope are obvious (e.g., it can cause name conflicts), and we'll deduct points for it.

Destructive versus nondestructive operations. It's OK to use destructive operators, if you're sure that they won't have any bad side-effects. For example, in my code for Project 2, Lisp's built-in `sort` function (which I used to reorder the fringe list) reordered the list destructively. This was OK because it didn't modify any of the nodes in the fringe—it just changed the order in which they appeared.

As an example of a bad side-effect, suppose I call

`(start-game` *boundary finish-line time-limit*`)`

and suppose your code does something like `(setf (first` *finish-line*`) 'garbage)`. The bad side-effect is that this redirects a pointer in my copy of the finish line.[1] I tested for this kind of thing in Project 2 (everyone's code passed the test!), and I'll test for it again in Project 3.

How to Measure CPU Time

There's a built-in Lisp function `(get-internal-run-time)` that returns the current running time as an integer. The time usually is measured in milliseconds; but this may differ from one Lisp implementation to another, and there is a built-in global variable called `internal-time-units-per-second` to let you know what the time unit is. Thus, we can use the following code to measure a function's running time in milliseconds:

```
(defun run-time-in-ms ()
   ;; Divide by units-per-second to get seconds, then
   ;; then multiply by 1000 to get milliseconds.
   (* 1000 (/ (get-internal-run-time)
              internal-time-units-per-second)))

(defun running-time (func arglist)
   ;; measure the running time of func on arglist
   (let ((start-time (run-time-in-ms)))
      (apply func arglist)
      (- (run-time-in-ms) start-time)))
```

By the way, the above code illustrates why Lisp's representation of fractions is useful: the time conversion in `cumulative-time-in-ms` has no roundoff error.

[1]In contrast, `(setf (first (copy-tree` *finish-line*`)) 'garbage)` would be OK, though I can't imagine why you'd ever want to do it.

Using Your Code to Play Racetrack

To play a racetrack game, a control program will call `start-game` once to set up the game, and will call `next-velocity` repeatedly to find out what moves to make. The objective is to reach the finish line without crashing, and with as few moves as possible.

Suppose t is the value of the *time-limit* parameter. Then the control program needs to move the racecar once every t milliseconds.

- If `start-game` returns within t milliseconds, then the game starts normally. But if it takes mt milliseconds where $m > 1$, then everything after the first t milliseconds will count toward your total number of moves in the game. Since your racecar hasn't actually started moving yet, these moves will be at velocity $(0, 0)$, i.e., the racecar will stay at its initial location for $\lceil m - 1 \rceil$ moves.
- If `next-velocity` returns a new velocity within t milliseconds, then the next move will be at the new velocity. But if `next-velocity` takes mt milliseconds where $m > 1$, then the control program will need to move the racecar $\lceil m - 1 \rceil$ times at the old velocity and then once at the new velocity. This will work as follows:

```
loop:
    1. Let $(x, y)$ be the racecar's current location, $(v, w)$ be its current velocity, and $n$
       be the number of moves it has made so far.
    2. Call next-velocity. Let $(v', w')$ be the velocity that it returns, $t'$ be the
       CPU time that it took, and $m = \lceil t'/t \rceil$. Then the racecar has a sequence of $m$
       moves to make: $m - 1$ moves at the old velocity $(v, w)$, and one move at the
       new velocity $(v', w')$.
    3. For $i = 1$ to $m$ do the following:
        – Add 1 to $n$, and perform the $i$'th move in the above sequence.
        – If the racecar hit the boundary of the racetrack, then return failure.
        – If the racecar reaches the finish line, then return $n$.
repeat
```

I've posted a simple version of the control program that you can use while developing your code. It's at

`http://www.cs.umd.edu/users/nau/cmsc421/p3-control.lisp`

When you submit your code, we'll test it in several racecar games, using a more sophisticated version of the control program.

Things to Think About

Running time. Allegro Common Lisp returns the running time in milliseconds; but you shouldn't expect it to be accurate to the millisecond. Every time I've called `(get-internal-run-time)` in Allegro, the number of milliseconds has always been a multiple of 10.

Safe and unsafe states. If the amount of time t available for your search is small, then you might not be able to search ahead all the way to the finish line. In this case, how are you going to decide which move is best? Among other things, you'll want to avoid moving to states that are *unsafe*; i.e., states from which a crash is inevitable. Here's one possible heuristic for deciding whether a state s is safe: if you start in state s and you decelerate the car to a complete stop as quickly as you can, will the car hit the wall before it stops?

Whether to cache the state space. During a racetrack game, the next call to `next-velocity` will probably need to search some of the same space that you're searching during the current call to `next-velocity`. Thus, it seems like it might be possible to cache some of the old state space (e.g., by storing it in some global variables) so that you don't need to regenerate it again. But unfortunately, I think it would be quite difficult to get this to work correctly; and even if you get it working correctly, I don't think it would enable you to find significantly better solutions. Thus I don't recommend it.

Extra Credit

We'll compute a performance measure for each program as follows. The top ten programs (i.e., the ones with the ten highest performance measures) will each get 10 points of extra credit.

- Program p *outperforms* program q on game g if either p finishes successfully and q doesn't, or both programs finish successfully and p takes fewer moves than q.
- Programs p and q perform *equally well* on game g if they both finish successfully in the same number of moves, or if neither finish successfully.
- Program p's score on game g is the number of programs that it outperforms, plus 1/2 the number of programs that perform equally well to p.
- Program p's *performance measure* is the sum of its scores on all of the games in our test suite.