ECE 592 Homework 5

Kudiyar Orazymbetov (korazym@ncsu.edu)
Nico Casale (ncasale@ncsu.edu)

November 28, 2017

Contents *(Note that the entries are links.)*

List of Figures

Listings

1 Rod Cutting Problem

1.1 Dynamic Programming Formulation

The dynamic programming can be done in two different ways: top down and bottom up. We will try to implement the formulation when we go from bottom to up. It means we will iteratively add values to the profitable revenue from bottom and take the length i which will maximize our one step ahead revenue. We will continue doing this until we finish the length.

1.2 Implementation

The code will be given below. For $n = 96$ and $n = 100$, we got 210 and 220 respectively.

1.3 Asymptotic Computational Complexity

The big O computational complexity of our dynamic programming function will be $O(n)$ as we will have two loops one goes 1 to n, second is a constant number.

1.4 Runtime Analysis

The figure below represents the runtime for `n = 1:1:100`. It is the result as we predicted. Time should should grow linearly as n increases.

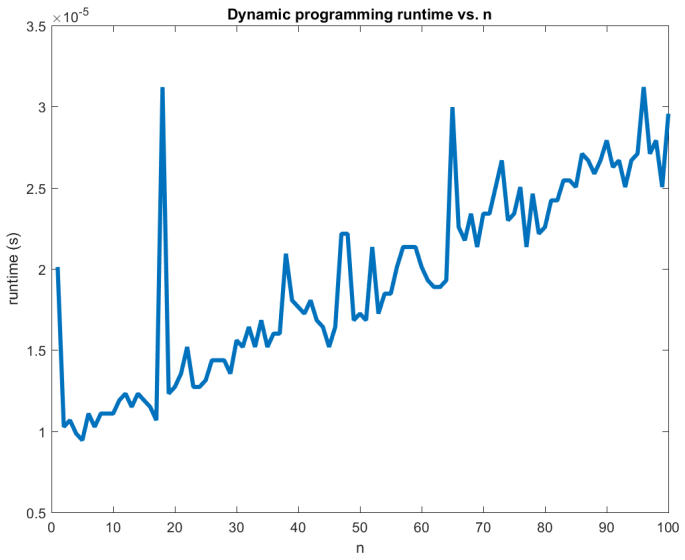

Figure 1.1: Dynamic programming runtime vs. n.

2 Comparison of Line Search and Golden Section Search

In finding the local or global extrema of a function, algorithms such as line search and golden section search can be very helpful. For this problem, we used the two algorithms to find the global minimum on the range $x = [0.01,\ 10]$ of the function

$$y = 3x + log(x) \tag{2.1}$$

The actual global minimum is achieved at approximately $x = 0.333$, as seen in the figure below.

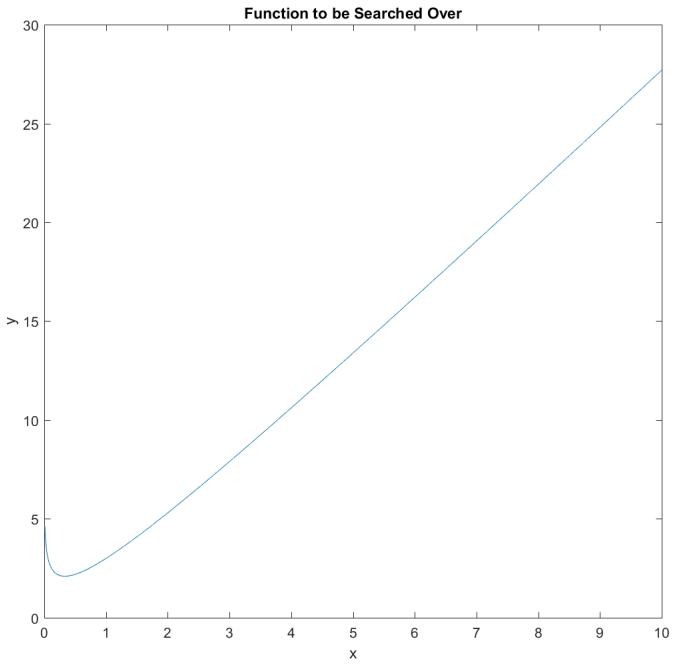

Figure 2.1: The function to be searched over in Problem 2.

2.1 Line Search

To implement line search, we modified the example code from the course webpage. We converted line search to be a function with the signature:

```
function minimum = line_search(a, b, num_iter)
```

`line_search(.)` returns the x-value of the minimum of the function, where `a` and `b` are the minimum and maximum of the range to be searched respectively. `num_iter` is the maximum number of iterations to run. This modular function allows us to easily time the line search algorithm in `pr2.m`.

2.2 Golden Section Search

Our implementation of line search was compared against an implementation of Golden Section Search[1]. Golden Section search is like line search in that it proceeds iteratively over a known function; however, its advantage over line search is that it is less computationally expensive. We will discuss this advantage in the next section. We implemented golden section search with the function signature

```
function minimum = golden(a, b, tol, num_iter)
```

[1]We referenced the following Wikipedia page for our implementation: https://en.wikipedia.org/wiki/Golden-section_search

`golden(.)` returns the x-value of the minimum of the function, where `a` and `b` are the minimum and maximum of the range to be searched respectively. `tol` is one of two stop conditions for the iterations of the function. If `num_iter` hasn't been reached and `(abs(c - d) < tol)`, then the function returns its minimum. This allows the user to specify a sort of accuracy, with a hard limit at `num_iter`. In our experiment, we were essentially only using the hard limit, as `tol` was set to $1e-10$. This modular function allows us to easily time the line search algorithm in `pr2.m`.

2.3 Performance Comparison

In `pr2.m`, we compare the speed of `line_search(.)` and `golden(.)` over 500 executions of the same calculation. This is to obtain a stable average time for each function. We also captured the shortest execution of the 100 repetitions to get an idea for the best-case performance. Finally, we captured the absolute difference in the minimum returned by each solution. This allowed us to compare the speed and accuracy of the two functions. `num_iter` was set to 10 for each algorithm. The results are illustrated in the table below.

	Min. Time (s)	Avg. Time (s)	Ret'd Min.
Line Search	0.00104	0.00315	0.3333
Golden Section Search	0.00017	0.00050	0.3445
Compare	6.092 × faster	6.296 × faster	0.0112 abs. diff. in accuracy

Table 1: First Comparison of Line Search and Golden Section Search Results.

We note that in 10 iterations, golden section search was faster but slightly less accurate. We ran the experiment again using 20 iterations for golden section search and 10 for line search. The results are in the table below.

	Min. Time (s)	Avg. Time (s)	Ret'd Min.
Line Search	0.001	0.00328	0.3333
Golden Section Search	0.00017	0.00043	0.3335
Compare	5.8 × faster	7.54 × faster	0.0002 abs. diff. in accuracy

Table 2: Second Comparison of Line Search and Golden Section Search Results.

In this case, since the iterations are not very expensive, and the function (2.1) is easy to calculate, we were able to attain a higher accuracy without a loss in speed. In fact, this experiment shows a speed increase over 500 repetitions of each function.

The advantage golden search has over line search is that it requires a less intensive search over the known function. Line search requires a sub-minimization over 10 (in this version) values of the function to choose the next locations for the forthcoming iteration. This is contrasted by the simplified computation in golden search, which requires only one update and two evaluations of the function to be minimized. The name *golden* originates in the way golden search chooses the next location to evaluate. As it turns out, the ratio between the values it chooses is the golden ratio, which is a classic value that shows itself in many mathematical and natural applications.

3 Code Listings

3.1 Main Code for Problem 2

Listing 1: Main code which will address parts a to d

```
%{
Nico
Cody
%}
%%
clear

p = [1 3 5 6 11];
%% n = 96, 100
pr96 = price2(96, p);
pr100 = price2(100, p);

%% plot timing
num_samps = 100;
times = zeros(1,num_samps);

for n = 1:num_samps

   tst = tic;
   pr = price2(n, p);
   times(n) = toc(tst);

   fprintf('pr = %d\n', pr);
end

f = instantiateFig(2);
plot(1:num_samps, times, 'LineWidth', 3);
prettyPictureFig(f);
xlabel('n');
ylabel('runtime (s)');
title('Dynamic programming runtime vs. n');
print('../images/pr2_timing','-dpng');
```

3.2 Main Code for Problem 2

Listing 2: Code to find best profit given length n

```
function pr = price2(n, p)
    final_value = zeros(1,n+1);
        %value(0) = 0;
    final_value(1) = 0;
    for i=1:n
        value = -inf;
        for j = 2:min(i+1,length(p)+1)
            value = max(value, p(j-1)+ final_value(i+2-j)); % j-1
        end
        final_value(i+1) = value;
    end
    pr = final_value(n+1);
```

```
end
```

3.3 Main Code for Problem 2

Listing 3: Code to measure performance of line search and golden section search.

```
%{
ECE 592 hw5

n casale
ncasale@ncsu.edu

kudiyar orazymbetov
korazym@ncsu.edu

Comparison of line search and golden section search
adapted from course code
17/11/26
%}

clear; close all;

addpath('utility');

f = instantiateFig(1);
x = 0.01:1e-4:10; % grid for visualizing signal
y = 3*x-log(x); % signal values
plot(x, y);
prettyPictureFig(f);
xlabel('x');
ylabel('y');
title('Function to be Searched Over');

print('../images/function', '-dpng');

REPS = 500;
num_iters = 10;
% run linesearch with speed measurement
tMinls = inf;
tic;
for rep = 1:REPS
   fprintf('ls rep = %d\n', rep);

   tSt = tic;
   minimumls = line_search(min(x), max(x), num_iters);
   tElapsed = toc(tSt);

   tMinls = min(tElapsed, tMinls);

end
tAvgls = toc/REPS;

% run golden section search with speed measurement
tMing = inf;
tic;
```

```
tol = 1e-10;
num_iters = 20;
for rep = 1:REPS
   fprintf('g rep = %d\n', rep);

   tSt = tic;
   minimumg = golden(min(x), max(x), tol, num_iters);
   tElapsed = toc(tSt);

   tMing = min(tElapsed, tMing);

end
tAvgg = toc/REPS;


% print results
fprintf('\nlinesearch:\n min = %2.5f,\n avg = %2.5f,\n fmin = %2.4f\n', ...
   tMinls, tAvgls, minimumls);
fprintf('\ngolden:\n min = %2.5f,\n avg = %2.5f,\n fmin = %2.4f\n', ...
   tMing, tAvgg, minimumg);
fprintf('\ncompare:\n min = %2.5f times faster,\n avg = %2.5f times faster,\n fmin = %2.4f abs
    diff in accuracy\n', ...
   tMinls/tMing, tAvgls/tAvgg, abs(minimumls-minimumg));
```