Numerical Python

optimization

CS101 Lecture #19

2016-11-28

Administrivia

Administrivia

- Homework #7 is due Monday, Dec. 4.

Administrivia

- Homework #7 is due Monday, Dec. 4.
- Mid-term II is on this Thursday
- Covers until (inclusive) today's lecture (lec11-19)

Warmup Question

Question #1

```
x = 'ABCD'
z = 'XYZ'

for a in itertools.product( x,y ):
    print( ' '.join( a ) )
```

Which of the following is *not* printed?

A `'A X'`

B `'B D'`

C `'C X'`

D `'D Z'`

Question #1

```
x = 'ABCD'
z = 'XYZ'

for a in itertools.product( x,y ):
    print( ' '.join( a ) )
```

Which of the following is *not* printed?

A `'A X'`

B `'B D'` $\star$

C `'C X'`

D `'D Z'`

Brute-Force Search

Brute-force search

- Brute-force search of a password:

```
def check_password( pwd ):
    if pwd == 'pas':
        return True
    else:
        return False

chars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'+\
        'abcdefghijklmnopqrstuvwxyz0123456789'
for pair in itertools.product( chars, repeat=3 ):
    pair = ''.join( pair )
    if check_password( pair ):
        print( pair )
```

Brute-force search

- Assume that a password can contain characters from the alphabet (upper- and lower-case); digits; and a selection of special characters (ampersand, dash): 86 characters.

Brute-force search

- Assume that a password can contain characters from the alphabet (upper- and lower-case); digits; and a selection of special characters (ampersand, dash): 86 characters.

Characters	Search Space
1	86
2	$86^2 = 7\,396$

Brute-force search

- Assume that a password can contain characters from the alphabet (upper- and lower-case); digits; and a selection of special characters (ampersand, dash): 86 characters.

Characters	Search Space
1	86
2	$86^2 = 7\,396$
3	$86^3 = 636\,056$
4	$86^4 = 54\,700\,816$
5	$86^5 = 4\,704\,270\,176$

Brute-force search

- Assume that a password can contain characters from the alphabet (upper- and lower-case); digits; and a selection of special characters (ampersand, dash): 86 characters.

Characters	Search Space
1	86
2	$86^2 = 7\,396$
3	$86^3 = 636\,056$
4	$86^4 = 54\,700\,816$
5	$86^5 = 4\,704\,270\,176$
10	$86^{10} = 2.2 \times 10^{19}$
20	$86^{20} = 4.9 \times 10^{38}$

Brute-force search

- If Python can try a password attempt every 1×10^{-7} s, how long does it take to crack a password of length *n*?

Characters	Search Space	Time
1	86	8.6×10^{-6} s
2	7 396	7.4×10^{-4} s
3	636 056	6.4×10^{-2} s
4	54 700 816	5.4 s
5	4 704 270 176	470.4 s

Brute-force search

- If Python can try a password attempt every 1×10^{-7} s, how long does it take to crack a password of length *n*?

Characters	Search Space	Time
1	86	8.6×10^{-6} s
2	$7\,396$	7.4×10^{-4} s
3	$636\,056$	6.4×10^{-2} s
4	$54\,700\,816$	5.4 s
5	$4\,704\,270\,176$	470.4 s
10	2.2×10^{19}	1.9×10^{14} s

Brute-force search

- If Python can try a password attempt every 1×10^{-7} s, how long does it take to crack a password of length *n*?

Characters	Search Space	Time
1	86	8.6×10^{-6} s
2	$7\,396$	7.4×10^{-4} s
3	$636\,056$	6.4×10^{-2} s
4	$54\,700\,816$	5.4 s
5	$4\,704\,270\,176$	470.4 s
10	2.2×10^{19}	1.9×10^{14} s
20	4.9×10^{38}	4.9×10^{31} s

Heuristic Optimization

Heuristic optimization

- In many cases, a "good-enough" solution is fine.

Heuristic optimization

- In many cases, a "good-enough" solution is fine.
- If we have measure of *relative* merit, we can assess candidate solutions by how good they are.

Heuristic optimization

- In many cases, a "good-enough" solution is fine.
- If we have measure of *relative* merit, we can assess candidate solutions by how good they are.
- Heuristic algorithms don't guarantee the 'best' solution, but are often adequate.

Hill-climbing algorithm

- **S**trategy: Always selecting neighboring candidate solution which improves on this one.

Hill-climbing algorithm

- **S**trategy: Always selecting neighboring candidate solution which improves on this one.
- **A**nalogy: Trying to find the highest hill by only taking a step uphill from where you are.

Hill-climbing algorithm

- **S**trategy: Always selecting neighboring candidate solution which improves on this one.
- **A**nalogy: Trying to find the highest hill by only taking a step uphill from where you are.
- **P**itfall: Get stuck at *local* optimum solution.

Steepest ascent algorithm

- **S**trategy: Tweaking our current solution by changing all elements to improve the result. Picking the candidate solution with the greatest improvement.

Steepest ascent algorithm

- **S**trategy: Tweaking our current solution by changing all elements to improve the result. Picking the candidate solution with the greatest improvement.
- **A**nalogy: Trying to find the highest hill by always taking the *steepest* step uphill from where you are.

Steepest ascent algorithm

- **S**trategy: Tweaking our current solution by changing all elements to improve the result. Picking the candidate solution with the greatest improvement.
- **A**nalogy: Trying to find the highest hill by always taking the *steepest* step uphill from where you are.
- **P**itfall: Finding a *local* optimum instead of the *global* optimum. QUESTION: how to find the steepest direction?

Random walk

- **S**trategy: Tweaking the current candidate solution at random, and possibly rejecting the solution if worse.

Random walk

- **S**trategy: Tweaking the current candidate solution at random, and possibly rejecting the solution if worse.
- **A**nalogy: Taking random steps near a hill, but maybe not taking the step if it's worse.

Random walk

- **S**trategy: Tweaking the current candidate solution at random, and possibly rejecting the solution if worse.
- **A**nalogy: Taking random steps near a hill, but maybe not taking the step if it's worse.
- **P**itfall: Converging slowly, can still miss best candidate solution. BENEFIT: has a way to get away from being stuck at local optima.

Example

- Let's revisit the bag-packing algorithm.

Example

- Our strategies:
  - Brute-force (last lecture)
  - Hill-climbing
    - Select most valuable item, then add next most valuable item, etc.
  - Random walk: make a random move from current solution, accept the move based on merit

Setup

```
import numpy as np
import matplotlib.pyplot as plt
import itertools

n = 10
items   = list( range( n ) )
weights = np.random.uniform( size=(n,) ) * 50
values  = np.random.uniform( size=(n,) ) * 100
```

Setup

```
def f( wts, vals ):
    total_weight = 0
    total_value = 0

    for i in range( len( wts ) ):
        total_weight += wts[ i ]
        total_value  += vals[ i ]

    if total_weight >= 50:
        return 0
    else:
        return total_value
```

Brute-force search

```
import itertools

max_value = 0.0
max_set = None
for i in range(n):
    for set in itertools.combinations( items,i ):
        wts  = []
        vals = []
        for item in set:
            wts.append( weights[ item ] )
            vals.append( values[ item ] )
        value = f( wts,vals )
        if value > max_value:
            max_value = value
            max_set = set
```

Tracking cases

```
max_value = 0.0
max_set = None
for i in range(n):
    for set in itertools.combinations( items,i ):
        wts  = []
        vals = []
        for item in set:
            wts.append( weights[ item ] )
            vals.append( values[ item ] )
        value = f( wts,vals )
        if value > max_value:
            max_value = value
            max_set = set
            print( max_value, vals )
```

Plot result

```
vals = values[np.array(max_set)]   #watch out this!
plt.plot( vals, 'bs' )
plt.xlim( ( 0, len(vals) ) )
plt.show()
```

Hill-climbing search

```
max_wt = 50.0

wts  = weights[ : ]
vals = values[ : ]

best_vals = [ ]
best_wts  = [ ]
best_vals.append( max( vals ) )
best_wts.append( wts[ vals.index( max( vals ) ) ] )
vals.remove( max( vals ) )
wts.remove( wts[ vals.index( max( vals ) ) ] )
```

Hill-climbing search

```
while sum(best_wts)+wts[vals.index(max(vals))] < max_wt:
    best_vals.append( max( vals ) )
    best_wts.append( wts[ vals.index( max( vals ) ) ] )
    vals.remove( max( vals ) )
    wts.remove( wts[ vals.index( max( vals ) ) ] )

# plot out values in final solution
plt.plot(best_vals, 'bs')
plt.xlim((0, len(best_vals)))
plt.show()
```

What is the strategy here?

- Pick the next item with the largest value until it overloads

What is the strategy here?

- Pick the next item with the largest value until it overloads
- QUESTION: is it the best strategy we can have?

What is the strategy here?

- Pick the next item with the largest value until it overloads
- QUESTION: is it the best strategy we can have?
  - Strategy 2: pick the next item with the least weight until it overloads

What is the strategy here?

- Pick the next item with the largest value until it overloads
- QUESTION: is it the best strategy we can have?
  - Strategy 2: pick the next item with the least weight until it overloads
  - Strategy 3: pick the next item with the highest value/weight ratio until overloads.

What is the strategy here?

- Pick the next item with the largest value until it overloads
- QUESTION: is it the best strategy we can have?
  - Strategy 2: pick the next item with the least weight until it overloads
  - Strategy 3: pick the next item with the highest value/weight ratio until overloads.
- Any of these guarantees to find the global optimal solution?

Random walk

```
#start with a configuration at random (empty set)
selected = np.zeros(n)
current_wts = weights[np.where(selected==1)]
current_vals = values[np.where(selected==1)]
#alter it at random with small likelihood of getting worse
for t in range( 1000 ):
    # make a change in 'selected'
    # two possible moves:  adding or swaping
    ...
    trial_wts = weights[np.where(selected==1)]
    trial_vals = values[np.where(selected==1)]
    if f(trial_wts,trial_vals)>f(current_wts,current_vals):
        #if improvement, accept the change
        current_wts,current_vals = trial_wts, trial_vals
    else:
        # do nothing
```

Random walk

```
#start with a configuration at random
selected = np.random.uniform(size=(10,)) < 0.2
current_wts = weights[np.where(selected==1)]
current_vals = values[np.where(selected==1)]
#alter it at random with small likelihood of getting worse
for t in range( 1000 ):
    # make a change in 'selected'
    # two possible moves:  adding or swaping
    ...
    trial_wts = weights[np.where(selected==1)]
    trial_vals = values[np.where(selected==1)]
    if f(trial_wts,trial_vals)>f(current_wts,current_vals):
        #if improvement, accept the change
        current_wts,current_vals = trial_wts, trial_vals
    else:
        #otherwise, accept the change with a *probability*
```

Random walk (extended)

- How to define the set of moves at each timestep?
- How to define the accept probability when there is no improvement?
  - *Simulated annealing*
  - *Monte Carlo Markov Chain* (MCMC)