Depth-First Search and Memoization

Fredo Durand & Ana Bell
MIT EECS 6.00
My Hobby:
Embedding NP-Complete Problems in Restaurant Orders

Chotchkie's Restaurant

<table>
<thead>
<tr>
<th>Appetizers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mixed Fruit</td>
<td>2.15</td>
</tr>
<tr>
<td>French Fries</td>
<td>2.75</td>
</tr>
<tr>
<td>Side Salad</td>
<td>3.35</td>
</tr>
<tr>
<td>Hot Wings</td>
<td>3.55</td>
</tr>
<tr>
<td>Mozzarella Sticks</td>
<td>4.20</td>
</tr>
<tr>
<td>Sampler Plate</td>
<td>5.80</td>
</tr>
</tbody>
</table>

Sandwiches

Barbecue | 6.55 |

We'd like exactly $15.05 worth of appetizers, please.

...Exactly? Uhh...

Here, these papers on the knapsack problem might help you out.

Listen, I have six other tables to get to — as fast as possible, of course. Want something on traveling salesman?
Backup your computer.

• Preferably online
• On a secondary drive
Optimization
Optimization problem

• Defined by two parts:
  – Objective function
  – Constraints

$ for knapsack
max capacity
Examples of optimization

http://www.sweetmaps.com/blog/?p=107
More optimization

http://www.astrokettle.com/

http://www.teachersdomain.org/resource/lsps07.sci.phys.energy.lightrefract/

Warehouse optimization

• bin packing
Optimization & stability

http://www.inf.ethz.ch/personal/whitinge/
http://people.csail.mit.edu/hishin/
def value(item):
    return item.getValue()

def weightInverse(item):
    return 1.0/item.getWeight()

def density(item):
    return item.getValue()/item.getWeight()

def greedy(Items, maxWeight, keyFcn):
    ItemsCopy = sorted(Items, key=keyFcn, reverse = True)
    result = []
    totalWeight = 0.0
    i = 0
    while totalWeight < maxWeight and i < len(Items):
        if (totalWeight + ItemsCopy[i].getWeight()) <= maxWeight:
            result.append((ItemsCopy[i]))
            totalWeight += ItemsCopy[i].getWeight()
        i += 1
    return result
Results

Items to choose from:
<clock, 175.0, 10.0>
<painting, 90.0, 9.0>
<radio, 20.0, 4.0>
<vase, 50.0, 2.0>
<book, 10.0, 1.0>
<computer, 200.0, 20.0>

Use greedy by VALUE to fill a knapsack of size $\text{maxWeight}= 20$
Total value of items taken = 200.0
<computer, 200.0, 20.0>

Use greedy by WEIGHT to fill a knapsack of size $\text{maxWeight}= 20$
Total value of items taken = 170.0
<book, 10.0, 1.0>
<vase, 50.0, 2.0>
<radio, 20.0, 4.0>
<painting, 90.0, 9.0>

Use greedy by DENSITY to fill a knapsack of size $\text{maxWeight}= 20$
Total value of items taken = 255.0
<vase, 50.0, 2.0>
<clock, 175.0, 10.0>
<book, 10.0, 1.0>
<radio, 20.0, 4.0>
Disclaimer

• Greedy has no guarantee of optimality
  – and usually is not optimal
• Which heuristic works best varies
def greedy(Items, maxWeight, keyFcn):
    ItemsCopy = sorted(Items, key=keyFcn, reverse=True)
    result = []
    totalWeight = 0.0
    i = 0
    while totalWeight < maxWeight and i < len(Items):
        if (totalWeight + ItemsCopy[i].getWeight()) <= maxWeight:
            result.append((ItemsCopy[i]))
            totalWeight += ItemsCopy[i].getWeight()
            i += 1
    return result
def greedy(Items, maxWeight, keyFcn):
    ItemsCopy = sorted(Items, key=keyFcn, reverse = True)
    result = []
    totalWeight = 0.0
    i = 0
    while totalWeight < maxWeight and i < len(Items):
        if (totalWeight + ItemsCopy[i].getWeight()) <= maxWeight:
            result.append((ItemsCopy[i]))
            totalWeight += ItemsCopy[i].getWeight()
        i += 1
    return result

• while loop: O(n)
• sort: O(n log n)
• total: O(n log n)
Bunny Backpack
Brute force depth
first search
Recursive exhaustive solution

Problem: input list of items

Base case: 0 item

Remove 1 item at a time

Sub problem:
Sub list without 1st item
Reduced capacity
Recursive exhaustive solution

decision about 1 item

- take it

  recursion 
  item \[ j \] 
  \( \text{capacity} - \text{weight/item} \)

- don't

  recursion 
  item \[ 1 \ldots \] 
  capacity

return total value + best list of items
Recursive exhaustive solution

- Take it
  - recursion
    - item (i), capacity - weight(item)
- Don't
  - recursion
    - item (i - 1), capacity

item

return total value + best list of items

return larger value + value(item) + best cost + item
Recursive exhaustive solution

- Either include first item or not
  - recursively explore whether to include other items
Recursive exhaustive solution

• High-level idea:

```python
def maxVal(toConsider, avail):
    '''toConsider: list of items; avail: weight capacity.
    returns max value and corresponding list of items'''
    #some corner\cases
    nextItem = toConsider[0]
    #Solve subproblem WITH nextItem and reduced capacity
    #Solve subproblem WITHOUT nextItem and same capacity
    #Return whichever one has better value
```
Recursive exhaustive solution

def maxVal(toConsider, avail):
    """toConsider: list of items; avail: weight capacity.
    returns max value and corresponding list of items""
    
    # some corner cases
    if toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0].getWeight() > avail:
        result = maxVal(toConsider[1:], avail)
    else:
        nextItem = toConsider[0]

        # Solve subproblem WITH nextItem and reduced capacity
        withVal, withToTake = maxVal(toConsider[1:], avail - nextItem.getWeight())
        withVal += nextItem.getValue()

        # Solve subproblem WITHOUT nextItem and same capacity
        withoutVal, withoutToTake = maxVal(toConsider[1:], avail)

        # Return whichever one has better value
        if withVal > withoutVal:
            result = (withVal, withToTake + [nextItem])
        else:
            result = (withoutVal, withoutToTake)

    return result
Result

Items Taken
<book, 10.0, 1.0>
<painting, 90.0, 9.0>
<clock, 175.0, 10.0>
Total value of items taken = 275.0

• Compare to greedy:

Items to choose from:
<clock, 175.0, 10.0>
<painting, 90.0, 9.0>
<radio, 20.0, 4.0>
<vase, 50.0, 2.0>
<book, 10.0, 1.0>
<computer, 200.0, 20.0>

Use greedy by VALUE to fill a knapsack of size maxWeight= 20
Total value of items taken = 200.0
<computer, 200.0, 20.0>

Use greedy by WEIGHT to fill a knapsack of size maxWeight= 20
Total value of items taken = 175.0
<book, 10.0, 1.0>
<vase, 50.0, 2.0>
<radio, 20.0, 4.0>
<painting, 90.0, 9.0>

Use greedy by DENSITY to fill a knapsack of size maxWeight= 20
Total value of items taken = 255.0
<vase, 50.0, 2.0>
<clock, 175.0, 10.0>
<book, 10.0, 1.0>
<radio, 20.0, 4.0>
Decisions tree: include item i or not

- 1st item?
  - Include
  - 2nd item?
    - Include
    - Don't
  - Don't include
    - 2nd item
      - Include
      - Don't
Decisions tree: include item i or not

- Full set of items: a, b, c, d

include a?
  - yes
  - no
    - include b?
      - yes
      - no
        - include c?
          - yes
          - no
            - ...
            - ...
    - include b?
      - yes
      - no
        - include c?
          - yes
          - no
            - ...
            - ...

...
Depth-first search

include a?

include b?

include c?

include c?

include b?

include c?

include c?
Depth-first search

breadth

include a?
  yes
  include b?
    yes
    include c?
    ... 
    no
  no
  include b?
    yes
    include c?
    ... 
    no
  include c?
  ...
More items?

• 10 items, max 40

```
Number of calls with brute force= 1931
Items Taken
<8, 9.0, 9.0>
<7, 6.0, 3.0>
<5, 9.0, 5.0>
<4, 5.0, 6.0>
<3, 8.0, 3.0>
<2, 5.0, 4.0>
<1, 4.0, 3.0>
<0, 8.0, 7.0>
Total value of items taken = 54.0
```

• 20 items, max 40

```
Number of calls with brute force= 332167
Items Taken
<18, 7.0, 4.0>
<17, 5.0, 1.0>
<14, 9.0, 9.0>
<12, 5.0, 1.0>
<11, 9.0, 7.0>
<7, 6.0, 3.0>
<5, 9.0, 5.0>
<3, 8.0, 3.0>
<2, 5.0, 4.0>
<1, 4.0, 3.0>
Total value of items taken = 67.0
```
More items?

• 30 items, max 40

Number of calls with brute force = 11224600
Items Taken
<29, 7.0, 5.0>
<25, 8.0, 5.0>
<21, 8.0, 3.0>
<18, 7.0, 4.0>
<17, 5.0, 1.0>
<12, 5.0, 1.0>
<11, 9.0, 7.0>
<7, 6.0, 3.0>
<5, 9.0, 5.0>
<3, 8.0, 3.0>
<1, 4.0, 3.0>
Total value of items taken = 76.0
Complexity
Memoization and Fibonacci
Naive recursive Fibonacci

\[ F(N) = F(N-1) + F(N-2) \]

\[ F(0) = F(1) = 1 \]
Naive recursive Fibonacci

• $F(n) = F(n-1) + F(n-2)$
• $F(0) = 0$
• $F(1) = 1$
Naive recursive Fibonacci

- \( F(n) = F(n-1) + F(n-2) \)
- \( F(0) = 0 \)
- \( F(1) = 1 \)

```python
def Fib(n):
    if n==0: return 0
    if n==1: return 1
    else:
        return Fib(n-1)+Fib(n-2)
```
def Fib(n):
    print n,
    if n==0: return 0
    if n==1: return 1
    else:
        return Fib(n-1)+Fib(n-2)
Recursive calls for Fib(10)

```python
def Fib(n):
    print(n,
    if n==0: return 0
    if n==1: return 1
    else:
        return Fib(n-1)+Fib(n-2)
```

<table>
<thead>
<tr>
<th>Depth</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Recursive tree for Fib(6)
Recursive tree for Fib(6)

Idea is remember answers we've solved before.
Redundancy
**Memoization**

```python
# Dictionary & memo = {'f': lambda N: 
    if N in memo: return memo[N]
    else:
        result = do the computation
        memo[N] = result
        return result
```
Memoization

• Store and reuse results

dictionary input => output
if result already in dictionary
  return it
else
  compute result
  store in dictionary
  return result

yes, the r is missing on purpose
Fibonacci with memoization

```python
def fastFib(n, memo={}):
    if n==0: return 0
    if n==1: return 1
    try:
        return memo[n]
    except KeyError:
        result=fastFib(n-1, memo)+fastFib(n-2, memo)
        memo[n]=result
        return result
```
Fibonacci with memoization

```python
def fastFib(n, memo={}):
    print n,
    if n==0: return 0
    if n==1: return 1
    try:
        return memo[n]
    except KeyError:
        result=fastFib(n-1, memo)+fastFib(n-2, memo)
        memo[n]=result
    return result
```

- Recursive calls:

```
10 9 8 7 6 5 4 3 2 1 0 1 2 3 4 5 6 7 8
55
```

- Recall brute force:

```
10 9 8 7 6 5 4 3 2 1 0 1 2 1 0 3 2 1 0 1 4 3 2 1 0 1 2 1 0 3 2 1 0 1 6 5
1 0 1 2 1 0 5 4 3 2 1 0 1 2 1 0 3 2 1 0 1 6 5
4 3 2 1 0 1 2 1 0 3 2 1 0 1 4 3 2 1 0 1 2 1 0
7 6 5 4 3 2 1 0 1 2 1 0 3 2 1 0 1 4 3 2 1 0 1
2 1 0 5 4 3 2 1 0 1 2 1 0 3 2 1 0 1 8 7 6 5 4
3 2 1 0 1 2 1 0 3 2 1 0 1 4 3 2 1 0 1 2 1 0 5
4 3 2 1 0 1 2 1 0 3 2 1 0 1 6 5 4 3 2 1 0 1 2
1 0 3 2 1 0 1 4 3 2 1 0 1 2 1 0
```
Tree for Fibonacci with memoization
Tree for Fibonacci with memoization