Chapter 15

Knapsack and Max-convolution

Copyright Oliver Serang, 2018
University of Montana Department of Computer Science

Where the subset-sum problem simply asks whether it is possible to reach total \( t \) using a subset of \( M \), the knapsack problem asks for the best possible way to reach total \( t \). We can think of this as allowing each of the menu items from Chapter 13 to also contribute some amount of “happiness” to your order, and seeks to maximize the food order \( S \subseteq M \) with \( \sum_{i \in S} = t \) that would yield the maximum happiness for your table:

\[
knapsack(M, t) = \max_{S \subseteq M: \sum_{i \in S} = t} \sum_{i \in S} \text{happiness}(i).
\]

15.1 Brute-force

Our brute-force approach to knapsack is quite similar to the brute-force approach to subset-sum: we will iterate through every subset \( S \) (by computing the power-set of \( M \)), and then of those that have total cost \( t \), we will take the one with maximum total happiness (Listing [15.1]).

Listing 15.1: Solving knapsack with brute force. The output is 14, indicating that the maximum happiness order for the table achieving total cost \( t = 120 \) yields a total happiness of 14.

```
import numpy
import itertools
```
# from stackoverflow:
def powerset(items):
    s = list(items)
    return itertools.chain.from_iterable(itertools.combinations(s, r) for r
    in range(len(s)+1))

def powerset_knapsack(M, t):
    max_happiness = -1
    for S in powerset(M):
        total_cost = sum(([ cost for cost, happiness in S ])
        total_happiness = sum(([ happiness for cost, happiness in S ])
        if total_cost == t and total_happiness > max_happiness:
            max_happiness = total_happiness

    if max_happiness == -1:
        print 'Warning: no subsets had sum', t, '(i.e., subset-sum of M, t
            was False)'
        return max_happiness

M_and_happiness = [(3,1), (5,10), (7,6), (9,2), (11,3), (13,8), (109,11),
    (207,4), (113,7), (300,18)]

print powerset_knapsack(M_and_happiness, 120)

15.2 Dynamic programming

Our dynamic programming approach to knapsack looks very similar to our
dynamic programming approach to subset-sum; the main difference is that
the \( n + 1 \)-partite graph for dynamic programming with subset-sum used
unweighted edges. Now, we will replace those with weighted edges. The
weight of each edge will correspond to the happiness contributed by that edge.
Edges that move horizontally (which exclude some \( m_i \)) will have weight +0,
whereas edges that move diagonally upwards will have weight +happiness(i).

Where our previous dynamic programming for subset-sum considered
only whether we could reach index \( t \) in the rightmost layer of the graph,
we now want to compute the highest-weighted path in the graph from the
far left node to the node at index \( t \) in the rightmost layer of the graph. For-
tunately, this is a fairly easy modification to make. Before, we assigned a
boolean to the nodes in the next layer using whether the previous layer was
reachable. As a set, the layer was written as
\[
layer_{\ell+1} = layer_{\ell} \oplus \{0, m_\ell\} \\
= \{i + j \mid \forall i \in layer_{\ell}, \forall j \in \{0, m_\ell\}\}.
\]

As we saw in Chapter 14, subset-sum dynamic programming can be written in terms of convolution between two arrays, which will count the number of ways to reach any node in the graph. When the number is greater than zero, then it is possible to reach a node:
\[
d_m = \sum_i a_i \cdot b_{m-i} \\
= a \odot b, \\
c_m = d_m > 0,
\]
where we use the array forms of the sets \(layer_{\ell}\) and \(\{0, m_\ell\}\):
\[
a_i = \begin{cases} 
1 & i \in layer_{\ell} \\
0 & i \not\in layer_{\ell}
\end{cases}, \\
b_i = \begin{cases} 
1 & i \in \{0, m_\ell\} \\
0 & i \not\in \{0, m_\ell\}
\end{cases}.
\]

Now, we will need to keep track of the best path reaching any node:
\[
d_m = \max_i a_i + b_{m-i} \\
c_m = d_m > 0.
\]

A dynamic programming implementation for knapsack is demonstrated in Listing 15.2. The runtime will match the dynamic programming approach to subset-sum from Chapter 14: \(r(n) \in \Theta(k \cdot n^2)\).

Listing 15.2: Solving knapsack with dynamic programming. As in the brute-force approach from Listing 15.1, the output is 14, indicating that the maximum happiness order for the table achieving total cost \(t = 120\) yields a total happiness of 14.

```python
def dynamic_programming_knapsack(M_and_happiness, t):
    # start current_layer[0] with "happiness" 0
```

The runtime will match the dynamic programming approach to subset-sum from Chapter 14: \(r(n) \in \Theta(k \cdot n^2)\).
current_layer = [0]

for m_i, happiness_i in M_and_happiness:
    next_layer_size = len(current_layer) + m_i
    # a value of -1 indicates the outcome is not possible:
    next_layer = [-1]*next_layer_size

    for j in xrange(len(current_layer)):
        if current_layer[j] != -1:
            # do not include m_i (add +0 to happiness):
            next_layer[j + 0] = max(next_layer[j + 0], current_layer[j] + 0)
            # include m_i (add +happiness_i to happiness):
            next_layer[j + m_i] = max(next_layer[j + m_i], current_layer[j] + happiness_i)

    current_layer = next_layer

# look in index t of current_layer and see whether or not it was
# reachable
return current_layer[t]

M_and_happiness = [(3,1), (5,10), (7,6), (9,2), (11,3), (13,8), (109,11),
                   (207,4), (113,7), (300,18)]

print dynamic_programming_knapsack(M_and_happiness, 120)

15.3 Generalized knapsack

Generalized knapsack is the knapsack-like version of generalized subset-sum:

$$\max\{happiness(0) + happiness(1) + \cdots + happiness(n-1) | m_0 + m_1 + \cdots + m_{n-1} = t\},$$

where $m_0 \in M_0, m_1 \in M_1, \ldots m_{n-1} \in M_{n-1}$, where $t \in T$, and where all $M_i \subseteq \{0,1,2,\ldots k - 1\}$. Generalized knapsack can be solved in a manner similar to generalize subset-sum. Just like the generalized subset-sum approach, the runtime of a dynamic programming approach to knapsack will be $r(n) \in \Theta(k^2 \cdot n^2)$. 
15.4 Max-convolution trees

Max-convolution trees can be used to solve the generalized knapsack problem; however, where merging a pair of nodes in a standard convolution tree is solved by standard convolution. Merging two vectors \( a \) and \( b \) to solve subset-sum was solved by convolution trees using the following:

\[
d_m = \sum_i a_i \cdot b_{m-i} = a \odot b,
\]

\[
c_m = d_m > 0,
\]

where \( a \odot b \) denotes the convolution between vectors \( a \) and \( b \). Using FFT, standard convolution has cost \( \in \Theta(\ell \log(\ell)) \), where \( \ell \) is the length of the longer vector between \( a \) and \( b \).

However, to solve knapsack, we no longer want to use standard convolution to merge nodes; instead, we want to use a type of “max-convolution”.

15.5 Max-convolution

Where standard convolution is defined using the operations \((+, \times)\),

\[
d_m = a \odot b = \sum_i a_i \cdot b_{m-i},
\]

(max, \(\times\)) convolution is defined as

\[
d_m = \max_i a_i \cdot b_{m-i}.
\]

Note that here we are using a product between elements \( a_i \cdot b_{m-i} \), whereas the max-convolution-like formula that occurs in the dynamic programming approach uses a sum between elements \( a_i + b_{m-i} \). If we can solve one of these efficiently, we can easily solve the other efficiently by simply transforming the vectors by using logarithms or exponentials, which will transform \( + \) operations into \( \times \) operations and vice-versa.

Unfortunately, the approach that we used with Karatsuba and FFT were only defined for rings, mathematical spaces that support inverse operations;
however, max-convolution uses a “semiring”: the max operation does not have an inverse operation. For example, if we have $z = x + y$, we can use only $z$ and $x$ to solve for $y$: $y = z - x$. On the other hand, if we know that $z = \max(x, y)$ we cannot necessarily use $z$ and $x$ to solve for $y$. Thus, the availability of fast convolution algorithms on these semirings is a hot and important question. Until 2006, no max-convolution algorithm $\in o(n^2)$ was known\(^1\). The algorithm published in 2006 is not substantially faster than naive: while it has a runtime $\in o(n^2)$, its runtime is $\notin O(n^{2-\epsilon})$.

15.6 Fast numeric max-convolution

Fast numeric max-convolution provides a numeric approximation for solving max-convolution. This enables us to use max-convolution trees without ever performing slow Cartesian products, as discussed in Chapter 14.

The numeric approach to max-convolution exploits the fact that the result at any index $m$ can be defined as a maximum over a vector $u^{(m)}$:

$$d_m = \max_i a_i \cdot b_{m-i}$$
$$= \max_i u_i^{(m)}$$
$$u^{(m)} = [a_i \cdot b_{m-i} | \forall i].$$

Note that we have not yet improved the performance over the naive $\Theta(n^2)$ approach; we have simply introduced a need to store a temporary vector, which may even slightly decrease the performance; however, we can start to think about mathematical approximations for estimating the maximum value in the $u^{(m)}$ vector.

The maximum value of $u^{(m)}$ can be found using the $\infty$-norm, also known as the Chebyshev norm:

$$\max_i u_i^{(m)} = \lim_{p \to \infty} \|u^{(m)}\|_p$$
$$= \lim_{p \to \infty} \left( \sum_i \left( u_i^{(m)} \right)^p \right)^{\frac{1}{p}}.$$

\(^1\)It was first discovered by Bremner et al. and published in 2006
This can be approximated using a numerically large value for $p$ rather than a limit as $p \to \infty$. We will denote this numerically large value as $p^*$:

$$
\max_i u_i^{(m)} \approx \left( \sum_i \left( u_i^{(m)} \right)^{p^*} \right)^{1/p^*}, \quad p^* \gg 1.
$$

If we expand this approximation back into the formula defining max-convolution, we get

$$
d_m = \max_i a_i \cdot b_m - i \\
\cong \left( \sum_i \left( u_i^{(m)} \right)^{p^*} \right)^{1/p^*} \\
= \left( \sum_i (a_i \cdot b_{m-i})^{p^*} \right)^{1/p^*} \\
= \left( \sum_i a_i^{p^*} \cdot b_{m-i}^{p^*} \right)^{1/p^*} \\
= (a^{p^*} \otimes b^{p^*})_m^{1/p^*} \\
d \approx (a^{p^*} \otimes b^{p^*})_m^{1/p^*}.
$$

Thus we see that we can simply take every element of $a$ to the power $p^*$, every element of $b$ to the power $p^*$, convolve using FFT convolution, and then take every element in the result to power $1/p^*$.

One interpretation for why this approximation works is that $z = x^{p^*} + y^{p^*}$ can behave both as a ring (allowing FFT convolution to be used) and as a semiring: Given $z$ and $x$, we can solve for $y$. But also, $z^{1/p^*} \approx \max(x, y)$ when $p^* \gg 1$.

When implementing this numerically, using a $p^*$ too large will result in numerically unstable behavior. This is because $(x + y) - x$ will numerically evaluate to 0 when $x \gg y$. Using double numbers, this will break down at some point when $\frac{y}{x} < \epsilon \approx 10^{-15}$ or so. Likewise, using a $p^*$ too small will result in a poor approximation of the Chebyshev norm. There are some sophisticated solutions, but one simple solution is to use a small exponential series of $p^*$ values. We know that the results will be roughly stable when the
result \( y = (x + y) - x \) is at least \( \epsilon \cdot x \). If we normalize the problems first so that the maximum values in the vectors \( a \) and \( b \) are both 1, then we are guaranteed a numerically stable result when the result at index \( m \) is \( > \epsilon \). Thus, we will perform a few convolutions using various \( p^* \) and then at each index in the result, choose the result from the largest \( p^* \) that was numerically stable. This numeric approach is demonstrated in Listing 15.3.

Listing 15.3: Numeric max-convolution. A numeric approach to estimating max-convolution. Where the exact max-convolution output by this example is 0.02, 0.06, 0.16, 0.2175, 0.58, 0.3625, 0.2, the fastest numeric approach yields result 0.02, 0.06000011, 0.1600363, 0.21778642, 0.58, 0.36540009, 0.2.

```python
import numpy
from scipy.signal import fftconvolve

def naive_convolve(x, y):
x_n = len(x)
y_n = len(y)
result_n = x_n+y_n-1

result = numpy.zeros(result_n)

for i in xrange(x_n):
    for j in xrange(y_n):
        result_index = i+j
        result[result_index] = result[result_index] + x[i]*y[j]
return result

def naive_max_convolve(x, y):
x_n = len(x)
y_n = len(y)
result_n = x_n+y_n-1

result = numpy.zeros(result_n)

for i in xrange(x_n):
    for j in xrange(y_n):
        result_index = i+j
        result[result_index] = max(result[result_index], x[i]*y[j])
return result

def numeric_max_convolve(x, y, log_p_max, epsilon=1e-10):
x_n = len(x)
y_n = len(y)
```
result_n = x_n+y_n-1
result = numpy.zeros(result_n)

all_p = 2.0**numpy.arange(log_p_max)
for p in all_p:
    result_for_p = fftconvolve(x**p, y**p)
    stable = result_for_p > epsilon
    result[stable] = result_for_p[stable]**(1.0/p)
return result

x=numpy.array([0.1, 0.3, 0.8, 0.5])
y=numpy.array([0.2, 0.15, 0.725, 0.4])

print 'Naive sum-convolution', naive_convolve(x,y)
print 'FFT sum-convolution', fftconvolve(x,y)
print
print
print 'Naive max-convolution', naive_max_convolve(x,y)
print 'FFT sum-convolution under p-norms:
for p in 2**numpy.arange(7.0):
    # call absolute value on each element using fabs so that very small
    # numeric errors like -1e-32 have real 1/p roots (otherwise results
    # are complex and numpy will complain):
    
    # nonetheless, for larger p, numeric instability creeps in:
    print ' p='+str(p), numpy.fabs(fftconvolve(x**p,y**p))**(1.0/p)
print
print
print 'Numeric max-convolution', numeric_max_convolve(x, y, 10)

Where the fastest exact approach to max-convolution is $\not\in O(n^{2-\varepsilon})$, this numeric approach reduces to a small number of FFT convolutions, each of which cost $\in \Theta(\ell \log(\ell))$. If we try an exponential sequence of $p^*$ values up to $p^* \leq p^*_{\text{max}}$ we are guaranteed that the $p^*$ used at each index will be at least half the maximum stable $p^*$. The runtime of this approach will be $\in \Theta(\ell \log(\ell) \log(p^*_{\text{max}}))$. Although this approach is not exact, it is by far the fastest approach in practice. For this reason, it is used in with max-convolution trees to estimate solutions to the generalized knapsack problem with runtime. When $p^*_{\text{max}}$ is a constant (and thus $\log(p^*_{\text{max}})$ is a constant
that can be removed from the runtime), the runtime for solving the general-
ialized knapsack problem with max-convolution trees and fast numerical
max-convolution will be $\in \Theta(k \cdot n \log(k \cdot n) \log(n))$. Thus we can see it is
at worst only a constant factor worse than the cost of standard convolution
tree from Chapter 14. This runtime is quasilinear, meaning it is only loga-
rithmically slower than the linear $k \cdot n$ cost of loading the data that defines
the problem.

Faster approaches, numeric or exact, for max-convolution are important
for several combinatoric problems.