# Knapsack Problem using Brute Force and Dynamic Programming

CSE 401: Artificial Intelligence

Utkarsh Gupta
A2305217557
7CSE 8Y

September 23rd, 2020

## 1 Knapsack Problem

**Implement a Knapsack problem using Brute Force Method and Dynamic Programming**

The knapsack problem is an optimization problem that takes a common computational need—finding the best use of limited resources given a finite set of usage options—and spins it into a fun story. A thief enters a shop with the intent to steal. He has a knapsack, and he is limited in what he can steal by the capacity of the knapsack. How does he figure out what to put into the knapsack?

### 1.1 Brute Force Approach

If we tried to solve this problem using a brute-force approach, we would look at every combination of items available to be put in the knapsack. For the mathematically inclined, this is known as a powerset, and a powerset of a set (in our case, the set of items) has $2^N$ different possible subsets, where $N$ is the number of items. Therefore, we would need to analyze $2^N$ combinations $(O(2^N))$. This is okay for a small number of items, but it is untenable for a large number. Any approach that solves a problem using an exponential number of steps is an approach we want to avoid.

[1]:
```
from itertools import product
from collections import namedtuple
try:
    from itertools import izip
except ImportError:
    izip = zip
```

[2]:
```
Reward = namedtuple('Reward', 'name value weight volume')

bagpack =   Reward('bagpack',  0, 25.0, 0.25)

items = [Reward('laptop',    3000,  0.3, 0.025),
         Reward('printer',   1800,  0.2, 0.015),
         Reward('headphone', 2500,  2.0, 0.002)]
```

[3]:
```python
def tot_value(items_count):
    """
    Given the count of each item in the sack return -1 if they can't be carried␣
 ↪or their total value.

    (also return the negative of the weight and the volume so taking the max of␣
 ↪a series of return
    values will minimise the weight if values tie, and minimise the volume if␣
 ↪values and weights tie).
    """
    global items, bagpack
    weight = sum(n * item.weight for n, item in izip(items_count, items))
    volume = sum(n * item.volume for n, item in izip(items_count, items))
    if weight <= bagpack.weight and volume <= bagpack.volume:
        return sum(n * item.value for n, item in izip(items_count, items)),␣
 ↪-weight, -volume
    else:
        return -1, 0, 0
```

[4]:
```python
def knapsack():
    global items, bagpack
    # find max of any one item
    max1 = [min(int(bagpack.weight // item.weight), int(bagpack.volume // item.
 ↪volume)) for item in items]

    # Try all combinations of reward items from 0 up to max1
    return max(product(*[range(n + 1) for n in max1]), key=tot_value)
```

[5]:
```python
import time

start = time.time()

max_items = knapsack()
maxvalue, max_weight, max_volume = tot_value(max_items)
max_weight = -max_weight
max_volume = -max_volume

print("The maximum value achievable (by exhaustive search) is %g." % maxvalue)
item_names = ", ".join(item.name for item in items)
print("  The number of %s items to achieve this is: %s, respectively." %␣
 ↪(item_names, max_items))
print("  The weight to carry is %.3g, and the volume used is %.3g." %␣
 ↪(max_weight, max_volume))

end = time.time()
print(f"\nThe total execution time taken is {end-start}.")
```

```
The maximum value achievable (by exhaustive search) is 54500.
  The number of laptop, printer, headphone items to achieve this is: (9, 0, 11),
respectively.
  The weight to carry is 24.7, and the volume used is 0.247.

The total execution time taken is 0.0247344970703125.
```

---

## 1.2 Dynamic Programming Approach

**Implement a Knapsack problem using Dynamic Programming. Compare the execution time of brute-force and dynamic programming algorithms.**

Instead, use a technique known as dynamic programming, which is similar in concept to memoization. Instead of solving a problem outright with a brute-force approach, in dynamic programming one solves subproblems that make up the larger problem, stores those results, and utilizes those stored results to solve the larger problem. As long as the capacity of the knapsack is considered in discrete steps, the problem can be solved with dynamic programming.

[6]:
```
from itertools import product
from collections import namedtuple
try:
    from itertools import izip
except ImportError:
    izip = zip
```

[7]:
```
Reward = namedtuple('Reward', 'name value weight volume')

bagpack =   Reward('bagpack',   0, 250, 250)

items = [Reward('laptop',    3000,   3, 25),
         Reward('printer',   1800,   2, 15),
         Reward('headphone', 2500,   20, 2)]
```

[8]:
```
def tot_value(items_count, items, bagpack):
    """
    Given the count of each item in the bagpack return -1 if they can't be␣
 ↪carried or their total value.

    (also return the negative of the weight and the volume so taking the max of␣
 ↪a series of return
    values will minimise the weight if values tie, and minimise the volume if␣
 ↪values and weights tie).
    """
```

```
    weight = sum(n * item.weight for n, item in izip(items_count, items))
    volume = sum(n * item.volume for n, item in izip(items_count, items))
    if weight <= bagpack.weight and volume <= bagpack.volume:
        return sum(n * item.value for n, item in izip(items_count, items)),␣
↪-weight, -volume
    else:
        return -1, 0, 0
```

[9]:
```
def knapsack(items, bagpack):
    table = [[0] * (bagpack.volume + 1) for i in range(bagpack.weight + 1)]

    for w in range(bagpack.weight + 1):
        for v in range(bagpack.volume + 1):
            for item in items:
                if w >= item.weight and v >= item.volume:
                    table[w][v] = max(table[w][v],
                                      table[w - item.weight][v - item.volume] +␣
↪item.value)

    result = [0] * len(items)
    w = bagpack.weight
    v = bagpack.volume
    while table[w][v]:
        aux = [table[w-item.weight][v-item.volume] + item.value for item in␣
↪items]
        i = aux.index(table[w][v])

        result[i] += 1
        w -= items[i].weight
        v -= items[i].volume

    return result
```

[10]:
```
import time

start = time.time()

max_items = knapsack(items, bagpack)
maxvalue, max_weight, max_volume = tot_value(max_items, items, bagpack)
max_weight = -max_weight
max_volume = -max_volume

print("The maximum value achievable (by exhaustive search) is %g." % maxvalue)
item_names = ", ".join(item.name for item in items)
print("  The number of %s items to achieve this is: %s, respectively." %␣
↪(item_names, max_items))
```

```
print("  The weight to carry is %.3g, and the volume used is %.3g." %␣
↪(max_weight, max_volume))

end = time.time()
print(f"\nThe total time taken is {end-start}.")
```

```
The maximum value achievable (by exhaustive search) is 54500.
  The number of laptop, printer, headphone items to achieve this is: [9, 0, 11],
respectively.
  The weight to carry is 247, and the volume used is 247.

The total time taken is 0.20116853713989258.
```

---