4810-1183: Approximation and Online Algorithms with Applications
Lecture Note 3: Approximation Algorithm for Knapsack Problem

Problem Definition

Suppose that you are in an all-you-can-eat strawberry farm, where you can have an unlimited amount of strawberry. Such type of farm is quite common in Japan, but, recently, many customers have a bad habit leaving a lot of remaining strawberry. Because of that, the association of all-you-can-eat strawberry farm issues a rule to prohibit leaving a partly-eaten strawberry. When you decide to eat a strawberry, you have to each a whole of it.

We have the following optimization model, because we want to eat the maximum amount of strawberry.

Input:	Positive integer n (number of strawberries) Positive real numbers $w_1, \ldots, w_n$ (weight of each strawberry) Positive real number W (maximum amount that we can eat strawberry) Assumption: $w_1 \le w_2 \le \cdots \le w_n$
Output:	Set $S \subseteq \{1, \ldots, n\}$ (set of strawberry we eat)
Constraint:	$\sum_{i \in S} w_i \le W$ (the weight sum of strawberry we eat is no more than W.)
Objective Function:	Maximize $\sum_{i \in S} w_i$ (maximize the amount of strawberry we eat.)

For example, when $n = 5, w_1 = 2, w_2 = 5, w_3 = 5, w_4 = 6, w_5 = 9$, $W = 10$, we should eat strawberry 2 and 3 to gain the maximum eating amount $w_2 + w_3 = 10 = W$. We attain the best output when $S = \{2,3\}$.

Corresponding to any particular output, the objective function gives us a value which we call "objective value". For example, the objective value of $S = \{1,2\}$ is $w_1 + w_2 = 7$. The best objective value we can get from a particular input is called "optimal value". We look for outputs with the objective value equal to the optimal value, and we will call those outputs as "optimal solutions". During this course, we will sometimes denote the optimal value by OPT, and sometimes denote the optimal solution by S^*.

The above problem is the simplified version of a problem called knapsack. The problem is known to be NP-hard even for the simplified version.

Algorithm

The following algorithm is for the simplified knapsack problem:

```
1:    $S \leftarrow \emptyset$
2:    For $j = 1$ to $n$
3:        If $\sum_{i \in S} w_i + w_j \le W$:
4:            $S \leftarrow S \cup \{j\}$
5:        Else:
6:            if $w_j \ge \sum_{i \in S} w_i$:
7:                $S \leftarrow \{j\}$
8:            break
```

The algorithm is mostly greedy algorithm. From the smallest to the heaviest, we pick strawberries to a bucket until the weight sum is more than we can eat. As, at the last step, we have more than we can eat in the bucket, we will remove some strawberry. We will remove the last strawberry when the weight of the last is smaller than that of the others combined. Otherwise, we will remove all but the last strawberry.

Recall the example in the previous section. We will choose strawberry 1,2, and 3 into our bucket. After we have strawberry 3, the weight sum ($w_1 + w_2 + w_3 = 12$) is more than $W = 10$. We have to choose between removing strawberry 3 or removing strawberry 1, 2. We will be able to eat more if we choose to remove strawberry 3, so we remain strawberry 1, 2 in our bucket. The output of the algorithm is $S = \{1,2\}$, and the objective value of the input is 7.

In this course, we will denote an input of an algorithm by S', and denote its objective value by SOL. S' is fairly nice, but it is not one of the optimal solutions as $SOL < OPT$. However, we can prove the following theorem.

Theorem 1: For any input, $SOL \geq 0.5 \cdot OPT$.

Proof: It is straightforward to show that $OPT \leq W$. We cannot eat more than W even in the optimal solution, as W is our maximum capacity.

In the algorithm, we will pick the strawberries until the weight sum is more than W. We know that

$$\text{Weight sum of other strawberries + weight of the last strawberry} \geq W.$$

By the inequality, we know that either "weight sum of other strawberries" or "weight of the last strawberry" is more than $0.5 \cdot W$. If both of them are smaller than $0.5 \cdot W$, the sum of them will not be more than W. Because of that, if we take the larger of the two, we will have the remaining weight larger than $0.5 \cdot W$. We have $SOL \geq 0.5 \cdot W \geq 0.5 \cdot OPT$. ■

Approximation Algorithm

Suppose that α is a positive real number less than 1. We will say that an algorithm is "an α-approximation algorithm" for a particular problem, if, for all inputs, we have

$$SOL \geq \alpha \cdot OPT.$$

When we want to find an output that maximize a value, OPT is the maximum objective value we can have from a particular input. Clearly, SOL, which is an arbitrary object value, cannot be larger than OPT. If we can solve the problem, we will have $SOL = OPT$ for all inputs. However, that is not possible when the problem is NP-hard. We unfortunately have $SOL < OPT$ for some input. Still, we can theoretically guarantee that SOL is always larger than 50% of OPT when $\alpha = 0.5$.

The guarantee can infer that, when we have an input with large optimal values, we are very likely to have a large object value from our algorithm. You might wonder why we have to care proving the inequality in the previous paragraph. The reason is, when you propose a method and you have only experiment results to support that the method is good, people cannot be sure if the results will be also good for their dataset or situations. They might not want to take risks implementing your methods, and your work might not be recognized as it should be. Because, with theories, people can easily know strong points and limitations of your methods, we do invite you to try having some guarantee for any of your proposals.

Approxability is one of the most common guarantee for an NP-hard problem [1]. We call α an "approximation ratio" of our algorithm.

Knapsack Problem

Previously, we want to maximize the strawberry weight we take. That is not something you usually want to maximize in practice. Some strawberries might be delicious than others, and we might want to eat them than just maximize the weight. Let suppose that we know our "happiness" from eating each strawberry in advance. We may formulize the problem into the following optimization model:

<u>Input</u>:	Positive integer n (number of strawberries) Positive real numbers $w_1, \dots, w_n$ (weight of each strawberry) Positive real number W (maximum amount that we can eat strawberry) ***Positive real number $h_1, \dots, h_n$ (happiness from eating each strawberry)*** ***Assumption:*** $\frac{h_1}{w_1} \geq \frac{h_2}{w_2} \geq \cdots \geq \frac{h_n}{w_n}$
<u>Output</u>:	Set $S \subseteq \{1, \dots, n\}$ (set of strawberry we eat)
<u>Constraint</u>:	$\sum_{i \in S} w_i \leq W$ (the weight sum of strawberry we eat is no more than W.)
<u>Objective Function</u>:	**Maximize $\sum_{i \in S} h_i$** **(maximize the happiness from the strawberries we eat.)**

The differences from the simplified version previously discussed are marked in bold italic. Instead of assuming that a smaller strawberry will come before a larger one, we sort the strawberries by the happiness gained per weight consumed. It is straightforward to show that the knapsack problem is NP-hard based on the fact that the simplified version is NP-hard.

We have the following algorithm for the knapsack problem.

```
1:  $S \leftarrow \emptyset$
2:  For $j = 1$ to $n$
3:      If $\sum_{i \in S} w_i + w_j \leq W$:
4:          $S \leftarrow S \cup \{j\}$
5:      Else:
6:          if $h_j \geq \sum_{i \in S} h_i$:
7:              $S \leftarrow \{j\}$
8:          break
```

The only difference from the previous algorithm in at Line 6. Previously, we chose to remove the last strawberry or the others based on the strawberrie s' weight. Now, we choose based on the strawberries' happiness. We will choose to remove the last strawberry if the happiness sum of the others are larger, and we will choose to remove the others if the happiness of the last strawberry is larger.

We can prove that the algorithm is 0.5-approximation algorithm, which mean that, for any particular input, the happiness we have from the algorithm is no less than 50% of the optimal. We will skip the proof for that in this course.

Bloom Filters

We will now move to an application of approximation algorithms to distributed computing. Consider the situation that we have a cache for a very dynamic network. There is a large number of data contained in the cache, and there are a very large number of inquiries if a particular data is there. We need a data structure that can answer to those inquiries in a few nanoseconds. If we use a sequential data structure like linked list, we might have to check all data in the worst case. That will take us too much searching time.

Suppose that all the possible data can be denoted by a positive integer less than 10^{80}. (The integer can be conversed from a 256-bit string, which represent the short description of each data.) We can have an array of booleans `a` with length 10^{80}. At the beginning, we set all `a[i]` to false. When we add an element `i` to the cache, we set `a[i]` to true. If there is an inquiry if `i′` is in the cache, we can immediately return `a[i′]`.

The method mentioned in the previous paragraph is very efficient. We can immediately update the array `a`, and the inquiry is answered almost immediately. However, it consumes a large amount of memory. We need 10^{80} bits for the data structure, and we are very unlikely to have to that large member.

We use hash functions to help us reduce the memory consumption. Suppose that the array size is m, which is much smaller 10^{80}. When we enter a particular information i to our cache, we will use the hash function f to map i to a random integer between 0 to $m-1$. Then, we will set `a[f(i)]` to `true`. When we have an inquiry if i' is in the cache, we will return `a[f(i′)]`.

We assume that, for every information i, $f(i)$ equals any integer between 0 to $m-1$ with probability $1/m$. Because of that, there will be about $10^{80}/m$ of information such that the hash of the information equals a particular number j. When `a[f(i)]=false`, we will be sure that we have not i in the cache. If i is put to the cache, `a[f(i)]`should have been set to true. On the other hand, if `a[f(i)]=true`, things might be more difficult. `a[f(i)]` might have been set to true because i' is entered to the cache, or it might be because of other $10^{80}/m$ information i' that have $f(i)=f(i')$.

Inspired by machine learning, we call the situation in the previous paragraph as false positive. Let us try to understand why the false positive is not good for a web cache system. In the system, users will search for a content with a particular URL in a cache. We usually have 2 layers of computations in the cache. The first layer will have a short description of a website, and answer if we have that website in the cache. Then, if there is, we will move to the second layer. We will search for the full web content and return the whole web content to users. We aim to design an algorithm for the first layer, which is called as "filter" here. When we frequently have false positive at the filter, we frequently have to move to the calculation at the second layer, which is much heavier. Thus, we do not want to have a lot of false positive, as it will decrease the efficiency of the cache system.

Let us try to calculate the probability of having a false positive. We will calculate a probability that the filter will return true, when we search for an information i that is not in the cache. Suppose that there are n different information in the cache, denoted by $i'_1, \dots, i'_n$. We will have `a[f(i)] = true` if and only if, for some k, we have $f(i)=f(i'_k)$. By contrapositive, we will have `a[f(i)] = false` if and only if, for all k, we have $f(i)\neq f(i'_k)$. Because the hash function f gives us a uniform random number, the probability that $f(i'_k)$ equals to a particular value is $\frac{1}{m}$. $f(i'_k)=f(i)$ with probability $\frac{1}{m}$. Thus, for a particular k, $f(i'_k)\neq f(i)$ with probability $1-\frac{1}{m}$. We can consider the event when $f(i'_k)\neq f(i)$ for each k as independent events. Because there is n possible values for k, there are n events with the same probability $1-\frac{1}{m}$. The probability of having all events occurred is $\left(1-\frac{1}{m}\right)^n$. By that, the probability of having `a[f(i)] = false` is $\left(1-\frac{1}{m}\right)^n$. The probability of having `a[f(i)] = true`, false positive, is $1-\left(1-\frac{1}{m}\right)^n$.

The probability of having false positive in the previous paragraph is usually very close to 1. There is a technique that help decreasing the probability called “bloom filter” [2]. Previously, we have just one hash function. Let us use p totally independent hash functions $f_1, \dots, f_p$. When we have a new information i in the case, we will set all `a[f1(i)], …, a[fp(i)]` to true .When there is an inquiry if i is in the cache, we will answer that i is in the cache if and only if `a[f1(i)], …, a[fp(i)]` are all true. We will answer that i is not in the cache otherwise.

We will have a false positive, only if all p bits are not correct. When p becomes larger, having all p bits incorrect are less likely to happen. On the other hand, when p becomes larger, we have to assign more bits to 1. We will have more incorrect bits by that. Here, we will calculate what is the best value of p.

Again, we will suppose that an information i is not in the cache. We want to calculate the probability that our filter will report that i is in the cache. Let us consider the probability that a particular `a[j]` is set to true. Because we have n different information in the cache and, when we have new information, we will assign true to p places, the event of assigning true happens for $n \cdot p$ times. Each event will not hit `a[j]` with probability $\left(1-\frac{1}{m}\right)$, so the probability of having all $n \cdot p$ events not hitting `a[j]` is $\left(1-\frac{1}{m}\right)^{np}$. For all j, `a[j]=false` with probability $\left(1-\frac{1}{m}\right)^{np}$, so `a[j]=true` with probability $1-\left(1-\frac{1}{m}\right)^{np}$. For each q, we have `a[fq(i)]=true` with probability $1-\left(1-\frac{1}{m}\right)^{np}$, and `a[f1(i)], …, a[fp(i)]` are all true with probability $\left(1-\left(1-\frac{1}{m}\right)^{np}\right)^p$. The probability of having false positive is $\left(1-\left(1-\frac{1}{m}\right)^{np}\right)^p$.

We can use calculus to find the best value of p, and it turns out that we will minimize the probability of having false positive when $p = \frac{m}{n} \cdot \ln 2$.

Adaptive Bloom Filter

Let us now consider the situation where we know the probability that each information i is in the cache. The probability could be predicted from a similar cache. We denote the probability of having i in the cache as P_i.

With different probability, the number of `a[j]` we will set to true for different information i is going to be different. Assume that we set p_i bits for an information i and the set of information in the cache is S. The number of events we set some random `a[j]` to true is $\sum_{i \in S} p_i$. The probability that an arbitrary `a[j]` is not set during those events is $\left(1-\frac{1}{m}\right)^{\sum_{i \in S} p_i}$. The authors of [2] have the following theorem:

<u>Theorem 2</u>: The probability of false positive is minimized, if the probability that `a[j]=true` is 0.5.

Because of that, to minimize the probability of false positive, we want to have $\left(1-\frac{1}{m}\right)^{\sum_{i \in S} p_i} = 0.5$. That is $\left(\sum_{i \in S} p_i\right) \cdot \ln\left(1-\frac{1}{m}\right) = \ln 0.5$. As we know that $1-x$ is very close to e^{-x} for a very small positive real number x, we have $1-\frac{1}{m} \approx e^{-\frac{1}{m}}$. Then,

$$\left(\sum_{i \in S} p_i\right) \cdot \ln\left(1-\frac{1}{m}\right) \approx \left(\sum_{i \in S} p_i\right) \cdot \ln\left(e^{-\frac{1}{m}}\right) = \left(\sum_{i \in S} p_i\right)\left(-\frac{1}{m}\right) = \ln 0.5.$$

If we negate both sides of the equation, we have

$$\frac{1}{m}\sum_{i \in S} p_i = -\ln 0.5 = \ln 2.$$

Because of the derivation, we want to have $\sum_{i \in S} p_i = m \cdot \ln 2$.

When we decide the value of p_i for each information, it is just a design step. We have not yet got an information in the cache, and we do not know what S is. Because of that, we cannot explicitly calculate the value of $\sum_{i \in S} p_i$. However, because we know that the probability of having $i \in S$ is P_i, we can estimate the value of $\sum_{i \in S} p_i$ by $\sum_i P_i \cdot p_i$. After a long discussion, we want to have the following equation.

$$\sum_i P_i \cdot p_i = m \cdot \ln 2.$$

Now, let us consider the probability of having false positive. Recall that, for each j, the probability of having `a[j]=true` is 0.5. Because, for an information i not in the cache, we have to check p_i bits, the probability of having all p_i bits true is $(0.5)^{p_i}$. When we assume that all information are inquired exactly one time, the expect number of false positive is $\sum_i (0.5)^{p_i}$. Because we want to optimize the number of false positive, we have the following optimization model:

Input:	For all information i, probability P_i (probability that we have i in S) positive integer m (cache size)
Output:	For all information i, positive integer p_i (number of random bits set to true when i arrives to cache)
Constraint:	$\sum_i P_i \cdot p_i = m \cdot \ln 2$.
Objective Function:	Minimize $\sum_i (0.5)^{p_i}$

Let us consider the objective function $\sum_i (0.5)^{p_i}$. We usually have a problem when the function is exponential of output, so we need something easier. Assume that the number of random bits set to true when a particular information, p_i, is not more than 3. Define 3 new variables $p_i^{(1)}, p_i^{(2)}$, and $p_i^{(3)}$ as follows:

$$p_i^{(j)} = \begin{cases} 0 & \text{if } p_i < j \\ 1 & \text{otherwise.} \end{cases}$$

By some calculation, when N is the number of possible information, we have

$$\begin{aligned}\sum_i (0.5)^{p_i} &= \sum_i \left(1 - 0.5 \cdot p_i^{(1)} - 0.25 \cdot p_i^{(2)} - 0.125 \cdot p_i^{(3)}\right) \\ &= N - \sum_i 0.5 \cdot p_i^{(1)} - \sum_i 0.25 \cdot p_i^{(2)} - \sum_i 0.125 \cdot p_i^{(3)}.\end{aligned}$$

As N does not depend on the output, minimizing $\left(N - \sum_i 0.5 \cdot p_i^{(1)} - \sum_i 0.25 \cdot p_i^{(2)} - \sum_i 0.125 \cdot p_i^{(3)}\right)$ is equivalent to minimizing $\left(-\sum_i 0.5 \cdot p_i^{(1)} - \sum_i 0.25 \cdot p_i^{(2)} - \sum_i 0.125 \cdot p_i^{(3)}\right)$. Also, as we know that minimizing $-x$ is equivalent to maximizing x, what we want to do is maximizing $\left(\sum_i 0.5 \cdot p_i^{(1)} + \sum_i 0.25 \cdot p_i^{(2)} + \sum_i 0.125 \cdot p_i^{(3)}\right)$.

Let $w_i^{(j)} = P_i$ for all i and j, and let $Q = \left\{p_i^{(j)}: p_i^{(j)} = 1\right\}$. Because $p_i = p_i^{(1)} + p_i^{(2)} + p_i^{(3)}$, we have

$$\begin{aligned}\sum_i P_i \cdot p_i = \sum_i P_i \cdot \left(p_i^{(1)} + p_i^{(2)} + p_i^{(3)}\right) &= \sum_i \left(P_i \cdot p_i^{(1)} + P_i \cdot p_i^{(2)} + P_i \cdot p_i^{(3)}\right) \\ &= \sum_i \left(P_i \cdot p_i^{(1)} + P_i \cdot p_i^{(2)} + P_i \cdot p_i^{(3)}\right) = \sum_i \left(w_i^{(1)} \cdot p_i^{(1)} + w_i^{(2)} \cdot p_i^{(2)} + w_i^{(3)} \cdot p_i^{(3)}\right) \\ &= \sum_{i,j} w_i^{(j)} p_i^{(j)} = \sum_{p_i^{(j)} \in Q} w_i^{(j)}.\end{aligned}$$

Let $h_i^{(1)} = 0.5, h_i^{(2)} = 0.25, h_i^{(3)} = 0.125$ for all i. We have

$$\begin{aligned}\left(\sum_i 0.5 \cdot p_i^{(1)} + \sum_i 0.25 \cdot p_i^{(2)} + \sum_i 0.125 \cdot p_i^{(3)}\right) &= \sum_i \left(h_i^{(1)} \cdot p_i^{(1)} + h_i^{(2)} \cdot p_i^{(2)} + h_i^{(3)} \cdot p_i^{(3)}\right) \\ &= \sum_{i,j} h_i^{(j)} p_i^{(j)} = \sum_{p_i^{(j)} \in Q} h_i^{(j)}.\end{aligned}$$

Let $W = m \cdot \ln 2$. We will have the following optimization model:

Input:	Positive integer N (number of possible information) Positive real numbers $w_i^{(j)}$ for $1 \le i \le N$ and $1 \le j \le 3$ Positive real number W Positive real number $\mathrm{h_i^{(j)}}$ for $1 \le \mathrm{i} \le \mathrm{N}$ and $1 \le \mathrm{j} \le 3$ Assumption: $\frac{\mathrm{h_1}}{\mathrm{w_1}} \ge \frac{\mathrm{h_2}}{\mathrm{w_2}} \ge \cdots \ge \frac{\mathrm{h_n}}{\mathrm{w_n}}$
Output:	Set $Q \subseteq \{p_i^{(j)}: 1 \le i \le N \text{ and } 1 \le j \le 3\}$
Constraint:	$\sum_{p_i^{(j)} \in Q} w_i^{(j)} = W$
Objective Function:	Maximize $\sum_{p_i^{(j)} \in Q} h_i^{(j)}$

In this paragraph, we will argue that it does not mater changing the constraint to $\sum_{p_i^{(j)} \in Q} w_i^{(j)} \le W$. We want to maximize $\sum_{p_i^{(j)} \in Q} h_i^{(j)}$, so we want to have elements in Q as much as possible. If $\sum_{p_i^{(j)} \in Q} w_i^{(j)} < W$, we may add more elements to Q to have a larger value of $\sum_{p_i^{(j)} \in Q} h_i^{(j)}$. Even we have the constraint $\sum_{p_i^{(j)} \in Q} w_i^{(j)} \le W$, it is very unlikely that we have $\sum_{p_i^{(j)} \in Q} w_i^{(j)} < W$. We are likely to have $\sum_{p_i^{(j)} \in Q} w_i^{(j)} = W$ anyway. Thus, it does not matter changing $\sum_{p_i^{(j)} \in Q} w_i^{(j)} = W$ to $\sum_{p_i^{(j)} \in Q} w_i^{(j)} \le W$.

By the previous paragraph, we will have the following optimization model.

Input:	Positive integer N (number of possible information) Positive real numbers $w_i^{(j)}$ for $1 \le i \le N$ and $1 \le j \le 3$ Positive real number W

Positive real number $h_i^{(j)}$ for $1 \leq i \leq N$ and $1 \leq j \leq 3$

Assumption: $\frac{h_1}{w_1} \geq \frac{h_2}{w_2} \geq \cdots \geq \frac{h_n}{w_n}$

Output: Set $Q \subseteq \{p_i^{(j)}: 1 \leq i \leq N \text{ and } 1 \leq j \leq 3\}$

Constraint: $\sum_{p_i^{(j)} \in Q} w_i^{(j)} \leq W$

Objective Function: Maximize $\sum_{p_i^{(j)} \in Q} h_i^{(j)}$

The optimization model is exactly same as the knapsack problem. We can use the 0.5-approximation algorithm for the knapsack problem to solve the optimization model. We can calculate all inputs of the optimization models from the properties of the Bloom filter, and receive the best number of bits for each information i from the output of the models

References

[1] D. P. Williamson and D. Shmoys, "*The design of approximation algorithms*", Cambridge University Press, 2011.

[2] M. Zhong, P. Lu, K. Shen, and J. Seiferas, "*Optimizing data popularity conscious bloom filters*", Proceedings of the 27th ACM Symposium on Principles of Distributed Computing (PODC'08), pages 355-364.