AI2615 算法设计与分析 2020-2021 春季学期

Lecture 7 – Some More Greedy Algorithms

2021 年 4 月 9 日

Lecturer: 张驰豪 *Scribe:* 陶表帅

In this lecture, we will study more greedy algorithms. We have learned Kruskal's algorithm for finding minimum spanning trees in the last lecture. We will learn another greedy algorithm for this problem: Prim's algorithm. Next, we will consider the problem of task assignment, and study a greedy algorithm for this. After that, we will study Huffman encoding, which again uses ideas from greedy algorithms. Finally, we will study the classical set cover problem. We will see that, although the greedy algorithm does not always output an optimal solution, it provides a solution that is reasonable close to the optimal one.

1 Prim's Algorithm

Recall the minimum spanning tree problem from the last lecture.

Problem 1 (Minimum Spanning Tree (MST)). Given a weighted undirected graph $G = (V, E, w)$ such that $w(e) > 0$ for any $e \in E$, find a connected subgraph $T = (V, E')$ containing all vertices of G with the minimum weight:

$$w(T) \triangleq \sum_{e \in E'} w(e).$$

A subgraph T with minimum weight is called a *minimum spanning tree*.

Kruskal's algorithm iteratively adds an edge that has minimum weight, while maintaining that the selected edges do not form any cycle. Prim's algorithm, on the other hand, works on vertices in a way that is similar to Dijkstra's algorithm.

Prim's algorithm starts by including an arbitrary vertex u_0 to a vertex set S. In each iteration, it finds the vertex in $V \setminus S$ that are closest to S, adds this vertex to S, and updates the distances for all the neighbors of this vertex. The algorithm is shown in Algorithm 1. The minimum spanning tree is stored by "pre" in the algorithm, where $\text{pre}(u)$ stores the parent of u in the tree.

The difference between Prim's algorithm and Dijkstra's algorithm is that, Dijkstra's algorithm maintains the distances of vertices to the source vertex s, while Prim's algorithm maintains the distances of vertices to the set S.

If the priority queue is implemented by a heap, it is easy to check that Prim's algorithm takes $O(m \log n)$ time.

Algorithm 1 Prim's algorithm

PRIM($G = (V, E, w)$)

```
for all u ∈ V:
    cost(u) ← ∞
    pre(u) ←NIL
endfor
Pick an initial vertex u_0 ∈ V
cost(u_0) ← 0
build a priority queue H = Priority_Queue(V) with respect to the costs
while H ≠ ∅:
    v ← H.delete_min()  // pop out a vertex with minimum cost
    for each {v, z} ∈ E:
        if cost(z) > w(v, z):
            cost(z) ← w(v, z)
            pre(z) ← v
        endif
    endfor
endwhile
```

2 Task Assignments

Problem 2 (Task Assignment). Given a set of n tasks such that each task i has an active interval $(s(i), f(i))$ where $s(i)$ is the starting time and $f(i)$ is the finishing time, find a maximum subset of tasks whose active intervals are non-overlapping.

We first discuss a few greedy algorithms that do not work (meaning that they do not always output optimal solutions).

Consider the greedy algorithm that iteratively selects a task with earliest starting time $s(i)$ that does not overlap with the already selected tasks. The following example shows that this algorithm does not always output the optimal solution. There is a task with starting time 0 and finishing time 10. There are five tasks with active intervals $(1,2), (3,4), (5,6), (7,8)$ and $(9,10)$. This greedy algorithm will first choose the task with starting time 0, which will make the remaining five tasks unavailable. A clearly better solution is to select the remaining five tasks without selecting the first task.

Consider the greedy algorithm that iteratively selects a task with shortest active interval that does not overlap with the already selected tasks. This still does not work. Suppose there is a task with shortest active interval $(3,5)$ and there are two tasks with longer active intervals $(0,4)$ and $(4,8)$ that intersect $(3,5)$. This greedy algorithm will select the first task, while selecting the remaining two tasks is a better solution.

Consider the greedy algorithm that iteratively selects a task that minimizes the number of the tasks that intersect this task. This seems to be a better algorithm, but it still fails. We leave it as an exercise to find an example where this algorithm fails to output the optimal solution.

Here is a greedy algorithm that works: iteratively select a task with the earliest *finishing time* that does not overlap with the already selected tasks. The algorithm is formally described in Algorithm 2.

Algorithm 2 A greedy algorithm for task assignment

TaskAssign$\{(s(i), f(i)) \mid i = 1, \ldots, n\}$

1: sort the task by the finishing time $f(i)$
2: $S \leftarrow \emptyset$
3: $F \leftarrow -\infty$ // F records the finishing time for the previous task selected
4: **for** $i = 1, \ldots, n$:
5: **if** $s(i) \geq F$: $S \leftarrow S \cup \{i\}$;
6: $F \leftarrow f(i)$
7: **endfor**
8: **return** S

Theorem 3. *Algorithm 2 outputs an optimal solution.*

Proof. Let $S = \{s_1, \ldots, s_\ell\}$ be the output of Algorithm 2, and let $O = \{o_1, \ldots, o_m\}$ be the optimal solution. Suppose both S and O are sorted by ascending order of the finishing times. We aim to show that $\ell \geq m$. We will prove the following observation: for any $i \leq \min\{\ell, m\}$, we have $f(s_i) \leq f(o_i)$. Notice that this observation immediately implies the theorem: suppose this observation is true and we have $\ell < m$; then $f(s_\ell) \leq f(o_\ell)$, and the algorithm should have included $o_{\ell+1}$ to S as well.

This observation can be proved by induction. For the base step, we have $f(s_1) \leq f(o_1)$, as the first task selected has the earliest finishing time. For the inductive step, suppose $f(s_i) \leq f(o_i)$. The task o_{i+1} satisfies $s(o_{i+1}) \geq f(o_i) \geq f(s_i)$, for otherwise there is an overlap between the active intervals for o_i and o_{i+1}, and O should not have been a valid solution. This implies that o_{i+1} is a candidate that is available to be included in $S = \{s_1, \ldots, s_i\}$. Since the algorithm always selects an available task with the earliest finishing time, we have $f(s_{i+1}) \leq f(o_{i+1})$. □

The time complexity for Algorithm 2 is dominated by the sorting step at Line 1. The overall time complexity is $O(n \log n)$.

3 Huffman Encoding

Suppose we want to compress an e-book whose characters are from a set $\mathcal{X}$ (for example, if this is an English book, $\mathcal{X}$ could be the set of 26 English letters). We would like to encode each character by a binary string.

That is, we want to construct a function $f : \mathcal{X} \to \{0,1\}^*$, such that, the compressed e-book encoded by f is as short as possible, and it is possible to recover the original content of the book from the compressed binary string.

For a toy example, consider $\mathcal{X}$ that contains only four alphabets: A, B, C, D. A naïve way to encode $\mathcal{X}$ is to use two-bit strings with $f(A) = 00$, $f(B) = 01$, $f(C) = 10$ and $f(D) = 11$. This will encode a message of length m to a binary string of length $2m$. In general, for $|\mathcal{X}| = t$, encoding a message of length m requires $m \log_2 t$ bits. Without any prior knowledge to the message, this is clearly optimal. However, in many scenarios, we know the frequency for each alphabet. Still using our example $\mathcal{X} = \{A, B, C, D\}$, suppose a message of 130 millions characters such that the number of occurrences for each of the four alphabets is given in Table 1. Encoding this message using the above-mentioned encoding scheme will use 260 millions bits.

Alphabet	Number of Occurrences
A	70 millions
B	3 millions
C	20 millions
D	37 millions

Table 1: The number of occurrences for each alphabet in a message with four alphabets.

Consider a different encoding scheme with $f(A) = 0$, $f(B) = 100$, $f(C) = 101$ and $f(D) = 11$. In this scheme, A is encoded by 1 bit, D is encoded by 2 bits, and each of B and C is encoded by 3 bits. Using this scheme, the message is encoded by $70 \times 1 + 3 \times 3 + 20 \times 3 + 37 \times 2 = 213$ millions bits, which is shorter than 260 millions bits before. The idea here is to use shorter binary string to encode an alphabet that has more frequency.

However, we must be cautious that the encoding should not cause ambiguity, in order to make recovering original message possible. For example, the encoding scheme given by $f(A) = 0$, $f(B) = 01$, $f(C) = 11$ and $f(D) = 001$ is ambiguous: given a binary string 001, it may represent AB or D. The encoding scheme must be *prefix-free*. That is, the encoding of one alphabet must not be the prefix of the encoding of another alphabet.

A prefix-free encoding can be represented by a full binary tree. The alphabets correspond to the leafs of the tree. For each internal node, the edge connecting to its left child represent bit "0" and the edge connecting to its right child represent bit "1". For each alphabet, the bits on the path from the root to the leaf representing this alphabet corresponds to the encoding of this alphabet. Figure 1 gives an example of the tree representation for the encoding scheme $f(A) = 0$, $f(B) = 100$, $f(C) = 101$ and $f(D) = 11$ used earlier.

Given an alphabet set $\mathcal{X} = \{a_1, \ldots, a_t\}$, a message and a full binary tree T representing a prefix-free encoding scheme, let f_i be the number of occurrences of a_i and d_i be the depth of the leaf node corresponding to

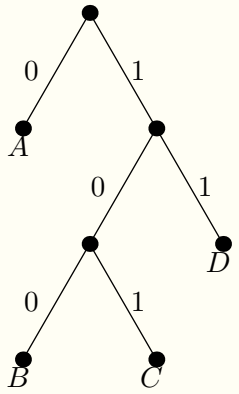

Figure 1: The full binary tree representation for the encoding scheme $f(A) = 0$, $f(B) = 100$, $f(C) = 101$ and $f(D) = 11$.

alphabet a_i. Notice that d_i is exactly the length of the binary string representing a_i. We define the cost of T as

$$\text{cost}(T) \triangleq \sum_{i=1}^{t} f_i \cdot d_i,$$

which represents the length of the binary string encoding the message. We would like to build the tree T that minimizes its cost.

The cost of T can be computed in the following alternative way. Using Figure 1 as an example, each occurrence of B in the message contributes 3 to the overall cost. An alternative way to consider this is to let B and each of B's ancestors, excluding the root node, contribute 1 to the overall cost. In this example, for each occurrence of B, node B contribute 1, the parent of B and C contributes 1, and the parent of D (which is an ancestor of B) contributes 1. Since B occurs 3 millions times, B is charged 3 millions, and each of B's ancestors (excluding the root) is charged 3 millions. Notice that each internal node is charged for all the leafs that are its descendants. For example, the parent of B and C is charged $3+20 = 23$ millions, and the parent of D is charged $3+20+37 = 60$ millions. Notice that the cost contributed by each internal node equals to the sum of the costs contributed by its two children. See Figure 2 for an example of this.

3.1 Huffman Encoding

Huffman encoding utilizes the idea of using shorter binary strings to represent alphabets with higher frequencies. Suppose $\mathcal{X} = \{a_1, \ldots, a_t\}$ is sorted by the ascending order of the frequencies, or the numbers of occurrences. Since a_1 and a_2 have least frequencies, we use the longest strings to encode it. On the full binary tree, we put the two nodes representing them at the deepest level. Since only the depths of the leafs matter, we can put the two nodes representing a_1 and a_2 under the same parent. Now we can treat this parent as a leaf node representing a new alphabet a_{12} whose number of occurrence is $f_1 + f_2$, and recursively solve the problem of tree building with $\mathcal{X}' = \{a_{12}, a_3, \ldots, a_t\}$. Of course, we need to sort

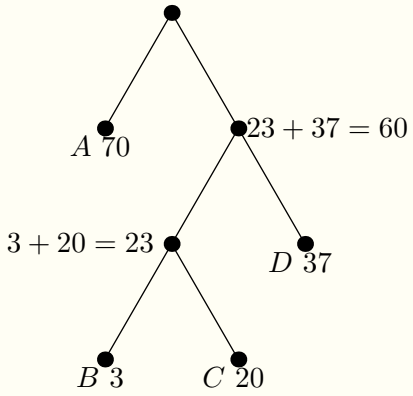

Figure 2: The cost contributed by each node for the encoding scheme $f(A) = 0$, $f(B) = 100$, $f(C) = 101$ and $f(D) = 11$ and the message described by Table 1.

$\mathcal{X}'$ again, as the frequency of a_{12} may be higher than a_3. This is precisely what Huffman encoding does. Notice that the idea behind Huffman encoding is still greedy: in each iteration, we always choose two alphabets with least frequencies. An example is given in Figure 3.

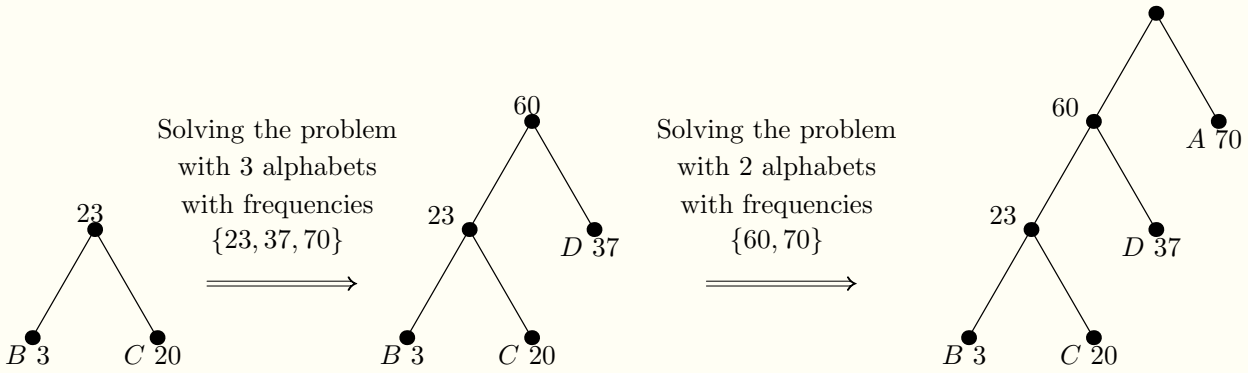

Figure 3: Example of Huffman encoding with the message described by Table 1.

Huffman encoding scheme is described in Algorithm 3. The time complexity for it is $O(t \log t)$.

3.2 Optimality of Huffman Encoding

Although Huffman encoding is not optimal in general, we will see a particular example where Huffman encoding is optimal.

Consider $\mathcal{X} = \{a_1, \ldots, a_t\}$ with $f_i = m \cdot 2^{-k_i}$, where m is the length of the message, and k_i is a positive integer. In addition, we make sure that $\sum_{i=1}^{t} 2^{-k_i} = 1$, so that $\sum_{i=1}^{t} f_i = m$. Notice that this is always possible:

$$f_1 = m \cdot 2^{-(t-1)}, f_2 = m \cdot 2^{-(t-1)}, f_3 = m \cdot 2^{-(t-2)}, f_4 = m \cdot 2^{-(t-3)}, f_5 = m \cdot 2^{-(t-4)}, \ldots, f_t = m \cdot 2^{-1}.$$

Algorithm 3 Huffman encoding scheme

HUFFMAN($\mathcal{X}, \mathbf{f} = \{f_1, \ldots, f_t\}$)

1: $H \leftarrow$ a priority queue of $\mathcal{X}$ with respect to $\mathbf{f}$
2: **for** $k = t+1, t+2, \ldots, 2t-1$:
3: $\quad i \leftarrow H$.delete_min()
4: $\quad j \leftarrow H$.delete_min()
5: $\quad \mathrm{pre}(i) = \mathrm{pre}(j) = k$ // build a node k and let i and j be its children
6: $\quad f(k) \leftarrow f(i) + f(j)$
7: $\quad H$.insert(k)
8: **endfor**

In the full binary tree T generated by Huffman encoding, alphabet a_i with $f_i = m \cdot 2^{-k_i}$ will be at the k_i-th level of the tree. In fact, all the nodes (internal nodes or leafs) at the i-th level must have cost $m \cdot 2^{-k_i}$. (Check this!) Therefore,

$$\mathrm{cost}(T) = \sum_{i=1}^{t} f_i k_i = m \sum_{i=1}^{t} 2^{-k_i} k_i = m \sum_{i=1}^{t} p_i \log_2 \left(\frac{1}{p_i}\right),$$

where $p_i \triangleq \mathbf{Pr}[X = a_i]$ is the probability that an alphabet X selected uniformly at random from the message is a_i.

In information theory, the *entropy* of a random variable X that takes value from $\{1, \ldots, t\}$ is defined by

$$\mathrm{Entropy}(X) = \sum_{i=1}^{t} \mathbf{Pr}[X = i] \log_2 \left(\frac{1}{\mathbf{Pr}[X = i]}\right).$$

In our case, we have

$$\mathrm{cost}(T) = m \cdot \mathrm{Entropy}(X).$$

Entropy measures the amount of information/uncertainty of a random variable. We can see that Huffman encoding is optimal in our example with $f_i = m \cdot 2^{-k_i}$, because each alphabet in the message contains at least $\mathrm{Entropy}(X)$ bits information.

To help better understanding entropy, let us look at some examples with messages that use alphabet set $\mathcal{X} = \{A, B\}$. If a message contains only A's, we know that such a message delivers 0 information. Indeed, the entropy is

$$p_A \cdot \log_2 \left(\frac{1}{p_A}\right) + p_B \cdot \log_2 \left(\frac{1}{p_B}\right) = 1 \cdot \log_2 1 + 0 \cdot \log_2 \left(\frac{1}{0}\right) = 0,$$

where $0 \cdot \log_2(\frac{1}{0}) = 0$ is based on $\lim_{x \to 0^+} x \cdot \log_2(\frac{1}{x}) = 0$.

Consider another extreme scenario that each alphabet of the message is equal likely to be A or B. In this case, the entropy is maximized:

$$p_A \cdot \log_2 \left(\frac{1}{p_A}\right) + p_B \cdot \log_2 \left(\frac{1}{p_B}\right) = \frac{1}{2} \cdot \log_2 2 + \frac{1}{2} \cdot \log_2 2 = 1.$$

In this case, we need exactly 1 bit to represent an alphabet: for example, 0 for A and 1 for B.

Consider an intermediate scenario that the message contains exactly $m-1$ A's and 1 B. In this case, the entropy is

$$p_A \cdot \log_2\left(\frac{1}{p_A}\right) + p_B \cdot \log_2\left(\frac{1}{p_B}\right) = \frac{m-1}{m} \cdot \log_2\left(\frac{m}{m-1}\right) + \frac{1}{m} \cdot \log_2 m.$$

This is the average number of bits to encode a single alphabet. The total number of bits to encode the entire message is

$$m \cdot \left(\frac{m-1}{m} \cdot \log_2\left(\frac{m}{m-1}\right) + \frac{1}{m} \cdot \log_2 m\right) = (m-1)\log_2\left(\frac{m}{m-1}\right) + \log_2 m.$$

When $m \to \infty$, $\log_2 m$ grows significantly faster than $(m-1)\log_2\left(\frac{m}{m-1}\right)$ (Check this!), and the total number of bits is $\log_2 m$. This is intuitive. To encode this message, it suffices to give the location of B in the message, which requires $\log_2 m$ bits to represent.

In the general case where f_i may not be represented as $m \cdot 2^{-k_i}$, Huffman encoding may not be optimal. However, there are ways to adapt it and make it optimal.

4 Set Cover

Consider the following problem in Figure 4. There are eleven communities. We want to build many hospitals to serve the people in these communities. Each hospital can only be built in a community, and we require that the distance between each community and the closest hospital near it is less than 5 kilometers.

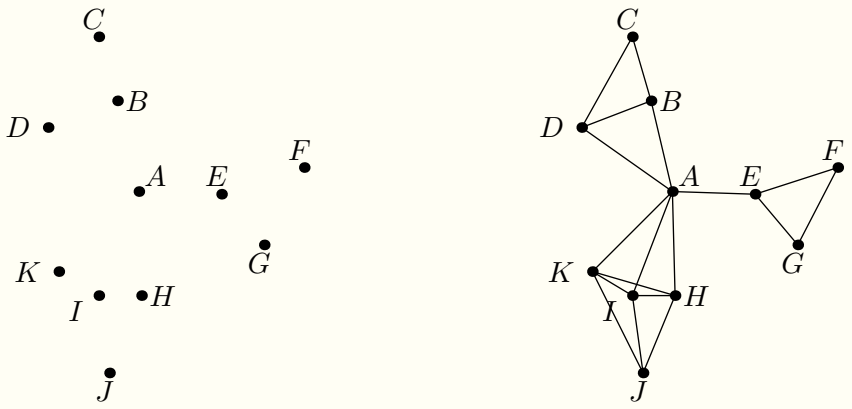

Figure 4: Eleven communities and communities that are within 5 kilometers of each other.

This is a typical *set cover* problem.

Problem 4 (Set Cover). Given a universe U of n elements and a collection of m subsets $S_1, \ldots, S_m \subseteq U$, find a minimum sub-collection of those subsets whose union is U.

The hospital building problem can be viewed by a set cover problem, where U contains all the communities and S_i is the set of all communities that are within distance 5 kilometers from community i.

There is a natural greedy algorithm for this problem: iteratively select a subset that contains the most uncovered elements, until all elements are covered by the chosen subsets. In the example in Figure 4, A will be chosen first, because it contains the most communities $\{A, B, D, E, H, I, K\}$. After that, the uncovered communities are C, F, G, J. The next hospital will be built on either F or G, each of which contains two uncovered communities: $\{F, G\}$. Finally, the algorithm will chose C and J in the next two iterations.
However, this solution is not optimal. We have built four hospitals, while a better solution is to build only three hospitals at B, E and I.
Nevertheless, we will show that this greedy algorithm is not too far away from being optimal.

Theorem 5. *Given a set cover instance, suppose the optimal solution requires choosing k subsets. The greedy algorithm will choose at most $k \ln n$ subsets.*

Proof. Let n_t be the number of the uncovered elements after t iterations of the algorithm. We have $n_0 = n$. After t iterations, we know that there exist k subsets that can cover all elements in U, and these k subsets can certainly cover all those n_t uncovered elements. By the pigeonhole-principle, there exists a subset that can cover at least n_t/k elements. Since the greedy algorithm selects a subset that contains the most uncovered elements, the next subset selected will contains at least n_t/k elements. Therefore, $n_{t+1} \le n_t - n_t/k$, which implies

$$n_t \le n_{t-1}\left(1 - \frac{1}{k}\right) \le n_{t-2}\left(1 - \frac{1}{k}\right)^2 \le \cdots \le n\left(1 - \frac{1}{k}\right)^t.$$

After $k \ln n$ iterations, the number of uncovered elements is at most

$$n\left(1 - \frac{1}{k}\right)^{k \ln n} < n\left(e^{-\frac{1}{k}}\right)^{k \ln n} = 1.$$

Since the number of uncovered elements needs to be an integer, all elements are covered after $k \ln n$ iterations. □

The greedy algorithm is a typical *approximation algorithm*. Based on Theorem 5, we say that the greedy algorithm achieves a $\ln n$-approximation (notice that the solution of the greedy algorithm is at most a $\ln n$ factor of the optimal solution).
A natural question is, can we do better in terms of approximation ratio? In fact, we will see in the future lectures that the set cover problem is NP-hard. Moreover, Dinur and Steurer [2014] showed that, unless P = NP, for any constant $\varepsilon > 0$, there does not exist a polynomial time algorithm that achieves a $(1-\varepsilon)\ln n$-approximation. We say that the set cover problem is NP-hard to approximate to within factor $(1-\varepsilon)\ln n$ for any constant $\varepsilon > 0$. This implies that the greedy algorithm is almost the best we can do.

参考文献

Irit Dinur and David Steurer. Analytical approach to parallel repetition. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 624–633, 2014. 9