Week 8

Greedy Algorithms

1. Greedy algorithms
2. Making change
3. Minimum spanning trees
4. Huffman Codes
General remarks

- We learn about **greedy algorithms**.

Reading from **CLRS** for week 8

1. Chapter 16.1-16.3
2. Chapter 23
Greedy Algorithms

This week and next week we will discuss two general methods which may be used to solve optimisation problems.

- The topic this week will be Greedy Algorithms.
- Next week we will discuss Dynamic Programming.

Idea of Greedy Algorithms

When we have a choice to make, make the one that looks best right now. Make a locally optimal choice in hope of getting a globally optimal solution.
The problem

From [Anany Levitin, Introduction to the Design and Analysis of Algorithms], Chapter 9, “Greedy Technique”, with some comments in square brackets:

\emph{Let us start with the change-making problem faced by millions of cashiers all over the world (at least subconsciously): give change for a specific amount \( n \) with the least number of coins of the denominations \( d_1 > d_2 > \cdots > d_m \) used in that locale. For example, the widely \( [?] \) used coin denomination in the United States \( [\text{of America}] \) are \( d_1 = 25 \) (quarter), \( d_2 = 10 \) (dime), \( d_3 = 5 \) (nickel), and \( d_4 = 1 \) (penny).}
The problem (cont.)

How would you give change with the coins of these denominations of, say, 48 cents? If you came up with the answer 1 quarter, 2 dimes, and 3 pennies \([48 = 25 + 2 \cdot 10 + 3]\), you followed [likely] — consciously or not — a logical [better “reasonable”] strategy of making a sequence of best choices among the currently available alternatives. Indeed, in the first step, you could have given one coin of any of the four denominations. “Greedy” thinking leads to giving one quarter because it reduces the remaining about the most, namely, to 23 cents. In the second step, you had the same coins at your disposal, but you could not give a quarter because it would have violated the problem’s constraints. So your best selection in this step was one dime, reducing the remaining amount to 13 cents. Giving one more dime left you with 3 cents to be given with three pennies.
Is this solution to the instance of the change-making problem optimal? Yes, it is. In fact, it is possible to prove that the greedy algorithm yields an optimal solution for every positive integer amount with these coin denominations. At the same time, it is easy to give an example of “weird” coin denominations — e.g., $d_1 = 7$, $d_2 = 5$, $d_3 = 1$ — which may not yield an optimal solution for some amounts. [The smallest amount for which the greedy strategy is not working is $n = 10$. The general problem is called the subset-sum problem, and is NP-complete.]
The approach applied in the opening paragraph to the change-making problem is called **greedy**. [...] The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step — and this is the central point of this technique — the choice made must be

**feasible**, *i.e.*, it has to satisfy the problem’s constraints.

**locally optimal**, *i.e.*, it has to be the [or one] best local choice among all feasible choices available on that step.

**irrevocable**, *i.e.*, once made, it cannot be changed on subsequent steps of the algorithm.
Reflection (cont.)

These requirements explain the technique’s name: on each step, it suggests a “greedy” grab of the best alternative available in the hope that a sequence of locally optimal choices will yield a (globally) optimal solution to the entire problem.
Making change (again)

**Input:** A list of integers representing coin denominations, plus another positive integer representing an amount of money.

**Output:** A minimal collection of coins of the given denominations which sum to the given amount.

**Greedy Strategy:** Repeatedly include in the solution the largest coin whose value doesn’t exceed the remaining amount.

**E.g.:** If the denominations are \((25,10,5,1)\) and the amount is 87, then

\[
87 = 25 + 25 + 25 + 10 + 1 + 1.
\]
Making change (again)

Input: A list of integers representing coin denominations, plus another positive integer representing an amount of money.

Output: A minimal collection of coins of the given denominations which sum to the given amount.

Greedy Strategy: Repeatedly include in the solution the largest coin whose value doesn’t exceed the remaining amount.

E.g.: If the denominations are (25,10,5,1) and the amount is 87, then

\[ 87 = 25 + 25 + 25 + 10 + 1 + 1. \]
Applicability of greedy algorithms

Greedy algorithms don’t always work.

For example, in Making Change, if the denominations were (25,11,5,1), then

\[ 15 = 11 + 1 + 1 + 1 + 1 = 5 + 5 + 5. \]

- However, quite often they do, or they come close enough to the optimal solution to make the outcome acceptable.
- This, and the fact that they are quite easy to implement, make them an attractive alternative for many hard problems.

Well-known instances of the use of Greedy Algorithms are the following problems:

- Minimum Spanning Trees (Kruskal’s Algorithm)
- Data Compression (Huffman Codes).
Remark on coin denominations

As far as I am aware, for all coin denominations in place today the greedy strategy works.

The only coin denomination in history I am aware of for which the greedy strategy doesn’t work was the British system before decimalisation (1971).

That system had

- half crown – 30 pence
- two shillings – 24 pence
- shilling – 12 pence
- sixpence – 6 pence
- threepence – 3 pence
- penny – 1 pence
- halfpenny – 1/2 pence.

Do you see an example where it doesn’t work?
Remark on coin denominations

As far as I am aware, for all coin denominations in place today the greedy strategy works.

- The only coin denomination in history I am aware of for which the greedy strategy doesn’t work was the British system before decimalisation (1971).

- That system had
  - half crown – 30 pence
  - two shillings – 24 pence
  - shilling – 12 pence
  - sixpence – 6 pence
  - threepence – 3 pence
  - penny – 1 pence
  - halfpenny – 1/2 pence.

- Do you see an example where it doesn’t work?

  \[ 48 = 30 + 12 + 6 = 2 \cdot 24. \]
Minimum Spanning Trees (MST)

Input: An undirected connected graph $G = (V, E)$, and a (positive) weight $w(e) \in \mathbb{R}_{\geq 0}$ on each edge $e \in E$.

Output: A subset $F \subseteq E$ of edges which connects all of the vertices $V$ of $G$, has no cycles, and minimises the total edge weight:

$$w(F) = \sum_{e \in F} w(e).$$

Example:
Minimum Spanning Trees (MST)

Input: An undirected connected graph \( G = (V, E) \), and a (positive) weight \( w(e) \in \mathbb{R}_{\geq 0} \) on each edge \( e \in E \).

Output: A subset \( F \subseteq E \) of edges which connects all of the vertices \( V \) of \( G \), has no cycles, and minimises the total edge weight:

\[
w(F) = \sum_{e \in F} w(e).
\]

Example:
Minimum Spanning Trees (MST)

Input: An undirected connected graph \( G = (V, E) \), and a (positive) weight \( w(e) \in \mathbb{R}_{\geq 0} \) on each edge \( e \in E \).

Output: A subset \( F \subseteq E \) of edges which connects all of the vertices \( V \) of \( G \), has no cycles, and minimises the total edge weight:

\[
w(F) = \sum_{e \in F} w(e).
\]

Example:
Kruskal’s algorithm

The following classic algorithm due to Kruskal is a greedy algorithm.

\[\text{KRUSKAL-MST}(G, w)\]

\[
\begin{align*}
&\text{// sort edges } E \text{ of } G \text{ by nondecreasing weight } w \\
1 &F = \emptyset \\
2 &\text{for each edge } e \in E \\
3 &\quad \text{if } F \cup \{e\} \text{ is acyclic} \\
4 &\quad \quad F = F \cup \{e\} \\
5 &\text{return } F
\end{align*}
\]

At each stage, the algorithm greedily chooses the cheapest edge and adds it to the partial solution \(F\), provided it satisfies the acyclicity criterion.

The running time of the algorithm is dominated by the sorting of edges in line 1 (assuming we can do the test in line 3 efficiently.) Hence, by using an optimal sorting algorithm, the running time of this algorithm is \(\Theta(E \lg E)\).
Kruskal’s algorithm

The following classic algorithm due to Kruskal is a greedy algorithm.

\texttt{Kruskal-MST}(G, w)

\hspace{1cm} // sort edges $E$ of $G$ by nondecreasing weight $w$
\begin{align*}
1 & \quad F = \emptyset \\
2 & \quad \text{for each edge } e \in E \\
3 & \quad \quad \text{if } F \cup \{e\} \text{ is acyclic} \\
4 & \quad \quad \quad F = F \cup \{e\} \\
5 & \quad \text{return } F
\end{align*}

At each stage, the algorithm greedily chooses the cheapest edge and adds it to the partial solution $F$, provided it satisfies the acyclicity criterion.

The running time of the algorithm is dominated by the sorting of edges in line 1 (assuming we can do the test in line 3 efficiently.) Hence, by using an optimal sorting algorithm, the running time of this algorithm is $\Theta(E \lg E)$. 
Kruskal’s algorithm

The following classic algorithm due to Kruskal is a greedy algorithm.

\[ \text{Kruskal-MST}(G, w) \]

\[
\begin{align*}
\text{// sort edges } E \text{ of } G \text{ by nondecreasing weight } w \\
1 & F = \emptyset \\
2 & \text{for each edge } e \in E \\
3 & \quad \text{if } F \cup \{e\} \text{ is acyclic} \\
4 & \quad F = F \cup \{e\} \\
5 & \text{return } F
\end{align*}
\]

At each stage, the algorithm greedily chooses the cheapest edge and adds it to the partial solution \( F \), provided it satisfies the acyclicity criterion.

The running time of the algorithm is dominated by the sorting of edges in line 1 (assuming we can do the test in line 3 efficiently.) Hence, by using an optimal sorting algorithm, the running time of this algorithm is \( \Theta(E \lg E) \).
Kruskal’s algorithm

The following classic algorithm due to Kruskal is a greedy algorithm.

Kruskal-MST($G$, $w$)

// sort edges $E$ of $G$ by nondecreasing weight $w$
1 $F = \emptyset$
2 for each edge $e \in E$
3     if $F \cup \{e\}$ is acyclic
4         $F = F \cup \{e\}$
5 return $F$

Kruskal’s Algorithm Illustrated

$w(a, f) = 1 \quad w(c, g) = 3 \quad w(e, g) = 4 \quad w(c, e) = 6$
$w(a, d) = 7 \quad w(d, f) = 7 \quad w(f, g) = 10 \quad w(d, e) = 11$
$w(b, d) = 12 \quad w(a, b) = 15 \quad w(b, c) = 20$
Kruskal’s algorithm illustrated

\[
\begin{align*}
  w(a, f) &= 1 \\
  w(a, d) &= 7 \\
  w(a, b) &= 15 \\
  w(b, d) &= 12 \\
  w(b, c) &= 20 \\
  w(c, g) &= 3 \\
  w(c, e) &= 6 \\
  w(d, f) &= 7 \\
  w(d, e) &= 11 \\
  w(e, g) &= 4 \\
  w(f, g) &= 10
\end{align*}
\]
Correctness of Kruskal’s algorithm

Fact: At any time during the execution of Kruskal’s Algorithm, \( F \) is a subset of some MST.

Proof: By induction on \( |F| \). This is clearly true (initially) when \( F = \emptyset \).

Suppose \( F \neq \emptyset \), and let \( e = (a, b) \) be the most recently added edge. Let \( T \) be a MST which (by induction) contains \( F - \{e\} \). Let \( f \) be the first edge on the (unique) path in \( T \) going from vertex \( a \) to vertex \( b \) which is not in \( F - \{e\} \), and assume that \( f \neq e \).

We must have \( w(e) \leq w(f) \) (for otherwise \( f \) would have been included in \( F \) rather than \( e \)), and hence we could get another MST by replacing \( f \) by \( e \) in \( T \). Hence \( F \) is a subset of a MST. \( \square \)

Corollary: Kruskal’s Algorithm computes a MST.
Kruskal’s algorithm revisited

Recall Kruskal’s Algorithm for computing Minimal Spanning Trees. We left open the problem of efficiently testing for the presence of cycles (line 3). This is catered for by maintaining the connected vertices as disjoint sets.
Kruskal’s algorithm revisited

Recall Kruskal’s Algorithm for computing Minimal Spanning Trees. We left open the problem of efficiently testing for the presence of cycles (line 3). This is catered for by maintaining the connected vertices as disjoint sets.

A reminder: Data Structures for Disjoint Sets

**Make-Set**(x) creates a new set whose only element is x.
**Find-Set**(x) returns the representative of the set containing x.
**Union**(x, y) combines the set $S_x$ containing x and the set $S_y$ containing y, forming a single new set $S$.

Theorem: Using the linked-list representation of disjoint sets with this weighted-union heuristic, a sequence of $m$ **Make-Set**, **Union** and **Find-Set** operations, $n$ of which are **Make-Set** operations, takes $O(m + n \log n)$ time.
Kruskal’s algorithm revisited

Recall Kruskal’s Algorithm for computing Minimal Spanning Trees. We left open the problem of efficiently testing for the presence of cycles (line 3). This is catered for by maintaining the connected vertices as disjoint sets.

**Kruskal-MST**(\(G, w\))

1. sort edges of \(G\) by nondecreasing weight \(w\)
2. \(F = \emptyset\)
3. for each vertex \(v\) of \(G\)
   
   MAKE-SET(\(v\))

4. for each edge \(e = (u, v)\) of \(G\)
5. \hspace{1em} if \(\text{FIND-SET}(u) \neq \text{FIND-SET}(v)\)
6. \hspace{2em} \(F = F \cup \{e\}\)
7. \hspace{1em} UNION(\(u, v\))
8. return \(F\)
When greedy algorithms work

Not every optimisation problem can be solved using a greedy algorithm. (For example, Making Change with a poor choice of coin denomination.)

There are two vital components to a problem which make a greedy algorithm appropriate:

**Greedy-choice property:** A globally optimal solution to the problem can be obtained by making a locally-optimal (greedy) choice.

(A greedy algorithm does not look ahead nor backtrack; hence a single bad choice, no matter how attractive it was when made, will lead to a suboptimal solution.)

**Optimal substructure property:** An optimal solution to the problem contains optimal solutions to subproblems.

(A greedy algorithm works by iteratively finding optimal solutions to these subproblems, having made its initial greedy choice.)
MST revisited

Greedy-choice property for MST: If $T$ is a MST, then $T$ contains the edge $e$ with the least weight. (Otherwise we could replace some edge in $T$ with $e$ and arrive at a better solution.)

Optimal substructure property for MST: If $T$ is a MST, then removing the edge $e$ with the least weight leaves two MSTs of smaller graphs. (Otherwise we could improve on $T$.)
Data compression

We wish to compress a text file (a string of characters) using a binary code of variable length, that is, a code in which the number of bits needed for the encoding may vary from character to character.

We restrict our attention to prefix codes, that is, codes in which no codeword is a prefix of any other codeword.

For example, the code

\[
\begin{align*}
\text{a} & \mapsto 0 \\
\text{b} & \mapsto 10 \\
\text{c} & \mapsto 11
\end{align*}
\]

is a prefix code, whereas the code

\[
\begin{align*}
\text{a} & \mapsto 0 \\
\text{b} & \mapsto 11 \\
\text{c} & \mapsto 111
\end{align*}
\]

is not.

(In the latter case, there is no way of telling whether 111111 encodes bbb or cc.)
Why variable-length prefix codes

It is wasteful to use a fixed-length code. For example, a fixed-length code for the three characters a, b and c would require (at least) two bits for every character. But by encoding a with only one bit, we save one bit for every occurrence of a.

(Presumably a occurs more often than either b or c; the more frequently-occurring characters should have shorter codes.)

Prefix codes allow easy encoding and decoding.

- To encode a text we simply replace each character with its code and concatenate them.
- To decode the text, we identify the initial codeword, translate it back to the original character, remove it from the encoded file, and repeat the decoding procedure.

E.g., using the above prefix code, the string 0110101011 uniquely corresponds to acabbc.
The tree representation of a code

Any prefix code is represented in an obvious way by a binary tree. For example, the code

a → 0  b → 101  c → 100  d → 111  e → 1101  f → 1100

is represented by the following binary tree.

Using the tree, it is easy to decode any encoded text:

101011111111011101111

is easily decoded, character-by-character, to

\[
\begin{array}{cccccccc}
101 & 0 & 111 & 111 & 1101 & 1101 & 111 \\
b & a & d & d & e & e & d
\end{array}
\]
A more efficient code

Using the previous code, the text baddeed is encoded by a bit string of length 21.

If instead we use the code

- a → 1011
- b → 100
- c → 10100
- d → 0
- e → 11
- f → 10101

then the text baddeed would instead be encoded as the 14-bit string 10010110011110, rather than as a 21-bit string.

The tree representation of this code is as follows.

Can we improve on this? No!
Computing the encoded length

Given a prefix code tree $T$, we can compute the number of bits used to encode a given text.

For alphabet $C$, let $f(c)$ be the frequency (number of occurrences) of the character $c \in C$ in the text, and let $d_T(c)$ be the depth of the leaf labelled $c$ in $T$ (that is, the length of the code for $c$).

Then the number of bits used to encode the text is

$$B(T) = \sum_{c \in C} f(c)d_T(c).$$

For example, the number of bits used to encode baddeed is:

$$B(T) = f(a)d_T(a) + \cdots + f(f)d_T(f)$$
$$= 1 \cdot 4 + 1 \cdot 3 + \cdots + 2 \cdot 2 + 0 \cdot 5$$
$$= 4 + 3 + 0 + 3 + 4 + 0 = 14.$$
The Data Compression Problem

**Input:** A set $C$ of characters (which possibly appear in some text to be compressed), along with a function $f : C \rightarrow \mathbb{N}$ giving the number of times each character appears.

**Output:** A binary code which provides an optimal compression of a text over $C$ with occurrences as given by $f$.

**A Greedy Algorithm:** Construct the prefix code tree bottom-up, starting with all characters as leaves, and successively merging the two lowest-frequency sub-trees. 

(Thus high-frequency characters are merged after the lower-frequency characters, giving them the tendency to end up higher up in the tree.)

This is the idea underlying **Huffman Codes**.

A generalisation of this problem uses *relative frequencies* $f : C \rightarrow \mathbb{Q}_{>0}$ — then we can use it for many similar texts.
Constructing a Huffman code

**Huffman**($C$)

1. $n = |C| ; \quad Q = C$
2. for $i = 1$ to $n - 1$
3. allocate a new node $z$
4. $z.left = x = \text{Extract-Min}(Q)$
5. $z.right = y = \text{Extract-Min}(Q)$
6. $z.freq = x.freq + y.freq$
7. $\text{Insert}(Q, z)$
8. return $\text{Extract-Min}(Q) // \text{return root of tree}$

$Q$ is a priority queue, keyed on $f$.

Assuming the queue is implemented as a binary heap, its initialisation (line 1) can be performed in $O(n)$ time; and each heap operation takes time $O(lg n)$ time.

Thus the total running time is $O(n \lg n)$. 
Huffman coding illustrated

(1) $f:5$ e:9 c:12 b:13 d:16 a:45

(2) $c:12$ b:13 14 d:16 a:45
   0 1
   f:5 e:9

(3) 14 d:16
   0 1
   f:5 e:9
   c:12 b:13

(4) 25 a:45
   0 1
   c:12 b:13

(5) a:45

(6) 100 a:45
    0 1
    25 30
    0 1
    14 d:16
    f:5 e:9
    c:12 b:13

Minimum spanning trees
Greedy algorithms
Making change
Huffman Codes
Correctness of Huffman’s algorithm I

Let:  
- $T$ represent an optimal prefix code;  
- $x, y \in C$ be the two least-frequent characters;  
- $a, b \in C$ be siblings with the longest codes;  
- $T'$ be $T$ with $x \leftrightarrow a$ and $y \leftrightarrow b$ exchanged.

Fact:  
$B(T) - B(T') \geq 0$, so $T'$ also represents an optimal prefix code.

Proof:  
$B(T) - B(T') = \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c)$  
$= f(x)d_T(x) + f(y)d_T(y) + f(a)d_T(a) + f(b)d_T(b)$  
$\quad - f(x)d_{T'}(x) - f(y)d_{T'}(y) - f(a)d_{T'}(a) - f(b)d_{T'}(b)$  
$= f(x)d_T(x) + f(y)d_T(y) + f(a)d_T(a) + f(b)d_T(b)$  
$\quad - f(x)d_T(a) - f(y)d_T(b) - f(a)d_T(x) - f(b)d_T(y)$  
$= (f(a) - f(x))(d_T(a) - d_T(x))$  
$\quad + (f(b) - f(y))(d_T(b) - d_T(y)) \geq 0. \; \square$
Correctness of Huffman’s algorithm II

Let:
- \( T \) represent an optimal prefix code;
- \( x, y \in C \) be siblings with parent \( z \);
- \( T' = T - \{x, y\} \); \( C' = C - \{x, y\} \cup \{z\} \); and \( f(z) = f(x) + f(y) \).

Fact: \( T' \) represents an optimal prefix code for \( C' \).

Proof: \( B(T) = B(T') + f(x) + f(y) \).
   Hence if there were a more optimal tree, then replacing \( x \) and \( y \) under \( z \) in this tree would provide a more optimal solution to the original problem. \( \square \)

Corollary: The algorithm is correct.