Intro to Analysis of Algorithms
Greedy
Chapter 2

Michael Soltys

CSU Channel Islands

[Ed: 4th, last updated: September 16, 2025]

MCST

Given a directed or undirected graph $G = (V, E)$ its adjacency matrix is a matrix A_G of size $n \times n$, where $n = |V|$, such that entry (i, j) is 1 if (i, j) is an edge in G, and it is 0 otherwise.

An adjacency matrix can be encoded as a string over $\{0, 1\}$.

That is, given A_G of size $n \times n$, let $s_G \in \{0, 1\}^{n^2}$, where s_G is simply the concatenation of the rows of A_G.

We can check directly from s_G if (i, j) is an edge by checking if position $(i - 1)n + j$ in s_G contains a 1.

Definitions:

- undirected graph
- path
- connected
- cycle / acyclic
- tree
- spanning tree

Every tree with n nodes has exactly $n-1$ edges.

Claim 1: Every tree has a leaf.

Proof: A leaf is by definition a node with less than 2 edges adjacent on it. If a graph does not have a leaf, then it has a cycle: pick any node, leave it by one of its edges, arrive at a new node ...

Claim 2: Every tree of n nodes has exactly $n-1$ edges.

Proof: By induction on n. BC: $n=1$ is trivial. Then consider a tree T of $n+1$ nodes; pick a leaf (it has one by Claim 1). Remove the leaf and its edge, and obtain a new tree T' (why is T' a tree?). Apply IH to T' and conclude T is a tree.

We are interested in finding a minimum cost spanning tree for G, assuming that each edge e is assigned a cost $c(e)$.

The understanding is that the costs are non-negative real number, i.e., each $c(e)$ is in $\mathbb{R}^+$.

The total cost $c(T)$ is the sum of the costs of the edges in T.

We say that T is a *minimum cost spanning tree (MCST)* for G if T is a spanning tree for G and given any spanning tree T' for G, $c(T) \leq c(T')$.

Encodings

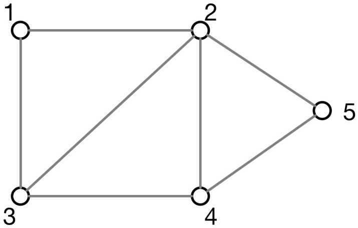
Difference between encoding and encryption. ASCII is an encoding; Caesar cipher is an encryption.

For example, the 7-bit word 1000001 represents (in ASCII) the letter 'A' and the word 0100110 represents '&'.

With 7 bits we can encode ...

Encodings are a *convention* for representing data. In Computer Science all data is eventually encoded as a string over the binary alphabet $\Sigma = \{0, 1\}$.

Encoding of a Graph



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Adjacency matrix

Encoded as a string:

0110010111110100110101010

Kruskal's Algorithm (A2.1)

Sort the edges: $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$
$T \longleftarrow \emptyset$
for $i : 1..m$ **do**
  if $T \cup \{e_i\}$ has no cycle **then**
    $T \longleftarrow T \cup \{e_i\}$
  end if
end for

But how do we test for a cycle, i.e., execute line 4 in the algorithm?

At the end of each iteration of the for-loop, the set T of edges divides the vertices V into a collection $V_1, \ldots, V_k$ of *connected components*.

That is, V is the disjoint union of $V_1, \ldots, V_k$, each V_i forms a connected graph using edges from T, and no edge in T connects V_i and V_j, if $i \neq j$.

A simple way to keep track of $V_1, \ldots, V_k$ is to use an array $D[i]$ where $D[i] = j$ if vertex $i \in V_j$.

Initialize D by setting $D[i] \longleftarrow i$ for every $i = 1, 2, \ldots, n$.

To check whether $e_i = (r, s)$ forms a cycle within T, it is enough to check whether $D[r] = D[s]$.

If e_i does not form a cycle within T, then we update:
$T \longleftarrow T \cup \{(r, s)\}$, and we merge the component $D[r]$ with $D[s]$ as shown in the algorithm in the next slide.

Merging Components (A2.2)

$k \longleftarrow D[r]$
$l \longleftarrow D[s]$
for $j : 1..n$ **do**
    if $D[j] = l$ **then**
        $D[j] \longleftarrow k$
    end if
end for

We now prove that Kruskal's algorithm works.

It is not immediately clear that Kruskal's algorithm yields a spanning tree, let alone a MCST.

To see that the resulting collection T of edges is a spanning tree for G, assuming that G is connected, we must show that (V, T) is connected and acyclic.

It is obvious that T is acyclic, because we never add an edge that results in a cycle.

To show that (V, T) is connected, we reason as follows. Let u and v be two distinct nodes in V.

Since G is connected, there is a path p connecting u and v in G. The algorithm considers each edge e_i of G in turn, and puts e_i in T *unless* $T \cup \{e_i\}$ forms a cycle.

But in the latter case, there must already be a path in T connecting the end points of e_i, so deleting e_i does not disconnect the graph.

This argument can be formalized by showing that the following statement is an invariant of the loop in Kruskal's algorithm:

The edge set $T \cup \{e_{i+1}, \ldots, e_m\}$ *connects all nodes in* V*.*

Promising

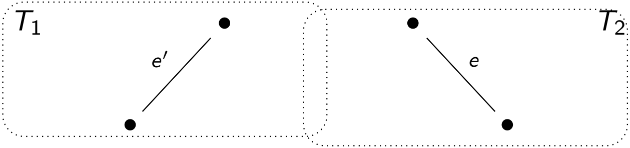
We say T is *promising* if it can be extended to a MCST with edges that have not been considered yet.

"T is promising"

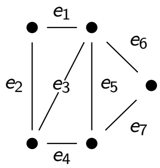
is a loop invariant of Kruskal's algorithm.

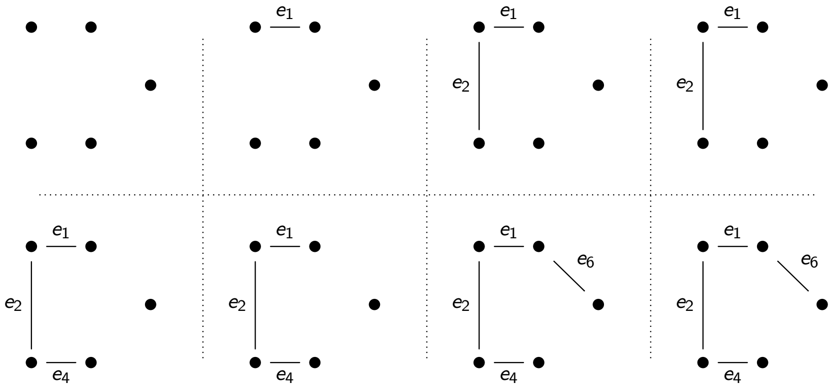
Exchange Lemma (Lemma 2.11)

Let G be a connected graph, and let T_1 and T_2 be any two spanning trees for G. For every edge e in $T_2 - T_1$ there is an edge e' in $T_1 - T_2$ such that $T_1 \cup \{e\} - \{e'\}$ is a spanning tree for G.



Example run





e1
e1
e2
e1
e2
e1
e2
e4
e1
e2
e4
e1
e6
e2
e4
e1
e6
e2
e4

Iteration	Edge	Current T	MCST extending T
0		$\emptyset$	$\{e_1, e_3, e_4, e_7\}$
1	e_1	$\{e_1\}$	$\{e_1, e_3, e_4, e_7\}$
2	e_2	$\{e_1, e_2\}$	$\{e_1, e_2, e_4, e_7\}$
3	e_3	$\{e_1, e_2\}$	$\{e_1, e_2, e_4, e_7\}$
4	e_4	$\{e_1, e_2, e_4\}$	$\{e_1, e_2, e_4, e_7\}$
5	e_5	$\{e_1, e_2, e_4\}$	$\{e_1, e_2, e_4, e_7\}$
6	e_6	$\{e_1, e_2, e_4, e_6\}$	$\{e_1, e_2, e_4, e_6\}$
7	e_7	$\{e_1, e_2, e_4, e_6\}$	$\{e_1, e_2, e_4, e_6\}$

We show the loop invariant.

Basis Case is easy.

Induction Step: assume T is promising; show it continues being promising after one more iteration of the loop.

Suppose edge e_i has been considered.

Case 1: e_i is rejected

Case 2: e_i is accepted. We must show $T \cup \{e_i\}$ is still promising.

We must show that $T \cup \{e_i\}$ is still promising. Since T is promising, there is a MCST T_1 such that $T \subseteq T_1$. We consider two subcases.

Subcase a: $e_i \in T_1$. Then obviously $T \cup \{e_i\}$ is promising.

Subcase b: $e_i \notin T_1$.

According to the Exchange Lemma, there is an edge e_j in $T_1 - T_2$, where T_2 is the spanning tree resulting from the algorithm, such that $T_3 = (T_1 \cup \{e_i\}) - \{e_j\}$ is a spanning tree.

Notice that $i < j$, since otherwise e_j would have been rejected from T and thus would form a cycle in T and so also in T_1.

Therefore $c(e_i) \leq c(e_j)$, so $c(T_3) \leq c(T_1)$, so T_3 must also be a MCST. Since $T \cup \{e_i\} \subseteq T_3$, it follows that $T \cup \{e_i\}$ is promising.

Jobs with deadlines and profits

n jobs and one processor

each job has a deadline and a profit, but all have duration 1

We think of a schedule S as consisting of a sequence of job "slots" $1, 2, 3, \ldots$, where $S(t)$ is the job scheduled in slot t.

A *schedule* is an array $S(1), S(2), \ldots, S(d)$ where $d = \max d_i$, that is, d is the latest deadline, beyond which no jobs can be scheduled.

If $S(t) = i$, then job i is scheduled at time t, $1 \leq t \leq d$.

If $S(t) = 0$, then no job is scheduled at time t.

A schedule S is *feasible* if it satisfies two conditions:

Condition 1: If $S(t) = i > 0$, then $t \leq d_i$, i.e., every scheduled job meets its deadline.

Condition 2: If $t_1 \neq t_2$ and also $S(t_1) \neq 0$, then $S(t_1) \neq S(t_2)$, i.e., each job is scheduled at most once.

Job Scheduling A2.3

Sort the jobs in non-increasing order of profits:
$g_1 \geq g_2 \geq \ldots \geq g_n$
$d \longleftarrow \max_i d_i$
for $t : 1..d$ **do**
    $S(t) \longleftarrow 0$
end for
for $i : 1..n$ **do**
    Find the largest t such that $S(t) = 0$ and $t \leq d_i$,
    $S(t) \longleftarrow i$
end for

A schedule is *promising* if it can be extended to an optimal schedule.

Schedule S' *extends* schedule S if for all $1 \leq t \leq d$, if $S(t) \neq 0$, then $S(t) = S'(t)$.

For example, $S' = (2, 0, 1, 0, 3)$ extends $S = (2, 0, 0, 0, 3)$.

We show by induction that S is promising is a loop invariant.

Basis case is easy

Induction step: Suppose that S is promising, and let S_{opt} be *some* optimal schedule that extends S.

Let S' be the result of one more iteration through the loop where job i is considered.

We must prove that S' continues being promising, so the goal is to show that there is an optimal schedule S'_{opt} that extends S'.

$S =$		0		0		j		
$S_{\mathrm{opt}} =$		0		i		j		

We consider two cases: job i can/cannot be scheduled

job i cannot be scheduled: easy

job i is scheduled at time t_0

job i is scheduled at time t_0, so $S'(t_0) = i$ (whereas $S(t_0) = 0$) and t_0 is the latest possible time for job i in the schedule S.

We have two subcases.

Subcase a: job i is scheduled in S_{opt} at time t_1:

If $t_1 = t_0$, then, as in case 1, just let $S'_{\text{opt}} = S_{\text{opt}}$.

If $t_1 < t_0$, then let S'_{opt} be S_{opt} except that we interchange t_0 and t_1, that is we let $S'_{\text{opt}}(t_0) = S_{\text{opt}}(t_1) = i$ and $S'_{\text{opt}}(t_1) = S_{\text{opt}}(t_0)$. Then S'_{opt} is feasible (why 1?), it extends S' (why 2?), and $P(S'_{\text{opt}}) = P(S_{\text{opt}})$ (why 3?).

The case $t_1 > t_0$ is not possible (why 4?).

Subcase b: job i is not scheduled in S_{opt}. Then we simply define S'_{opt} to be the same as S_{opt}, except $S'_{\mathrm{opt}}(t_0) = i$. Since S_{opt} is feasible, so is S'_{opt}, and since S'_{opt} extends S', we only have to show that $P(S'_{\mathrm{opt}}) = P(S_{\mathrm{opt}})$.

Claim: Let $S_{\mathrm{opt}}(t_0) = j$. Then $g_j \leq g_i$.

We prove the claim by contradiction: assume that $g_j > g_i$ (note that in this case $j \neq 0$). Then job j was considered before job i. Since job i was scheduled at time t_0, job j must have been scheduled at time $t_2 \neq t_0$ (we know that job j was scheduled in S since $S(t_0) = 0$, and $t_0 \leq d_j$, so there was a slot for job j, and therefore it was scheduled). But S_{opt} extends S, and $S(t_2) = j \neq S_{\mathrm{opt}}(t_2)$—contradiction.

Make Change A2.4

1. What would be the natural greedy alg for making change?
2. Does it work with all currencies?

Maximum weight matching

(Application to network switches.)

Let $G = (V_1 \cup V_2, E)$ be a bipartite, i.e, a graph with edge set $E \subseteq V_1 \times V_2$ with disjoint sets V_1 and V_2. $w : E \longrightarrow \mathbb{N}$ assigns a weight $w(e) \in \mathbb{N}$ to each edge $e \in E = \{e_1, \ldots, e_m\}$.

A *matching* for G is a subset $M \subseteq E$ such that no two edges in M share a common vertex. The weight of M is $w(M) = \sum_{e \in M} w(e)$.

What would be a natural Greedy alg?

Maximum weight matching

(Application to network switches.)

Let $G = (V_1 \cup V_2, E)$ be a bipartite, i.e, a graph with edge set $E \subseteq V_1 \times V_2$ with disjoint sets V_1 and V_2. $w : E \longrightarrow \mathbb{N}$ assigns a weight $w(e) \in \mathbb{N}$ to each edge $e \in E = \{e_1, \ldots, e_m\}$.

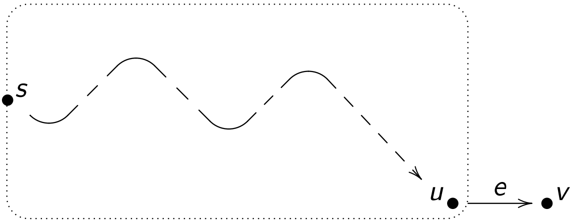
A *matching* for G is a subset $M \subseteq E$ such that no two edges in M share a common vertex. The weight of M is $w(M) = \sum_{e \in M} w(e)$.

What would be a natural Greedy alg?

See Problem 2.29 and Algorithm 2.6 given in its solution

Shortest path

Application to OSPF: Open Shortest Path First, see RFC 2328



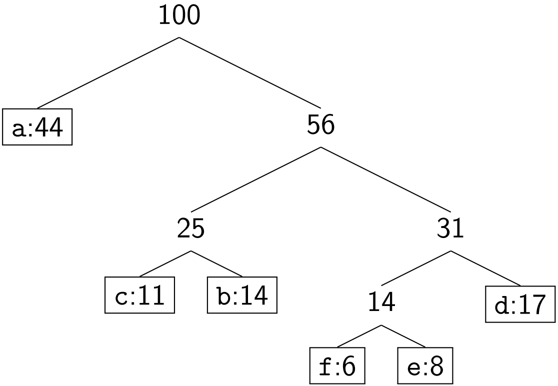
$$d'(v) = \min_{u \in S, e=(u,v)} d(u) + c(e).$$

Huffman Codes A2.5

Suppose that we have a string s over the alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}\}$, and $|s| = 100$.

Suppose also that the characters in s occur with the frequencies $44, 14, 11, 17, 8, 6$, respectively.

As there are six characters, if we were using fixed-length binary codewords to represent them we would require three bits, and so 300 characters to represent the string.


100
a:44
56
25
31
c:11
b:14
14
d:17
f:6
e:8