CSC3160: Design and Analysis of Algorithms

Week 3: Greedy Algorithms

Instructor: Shengyu Zhang

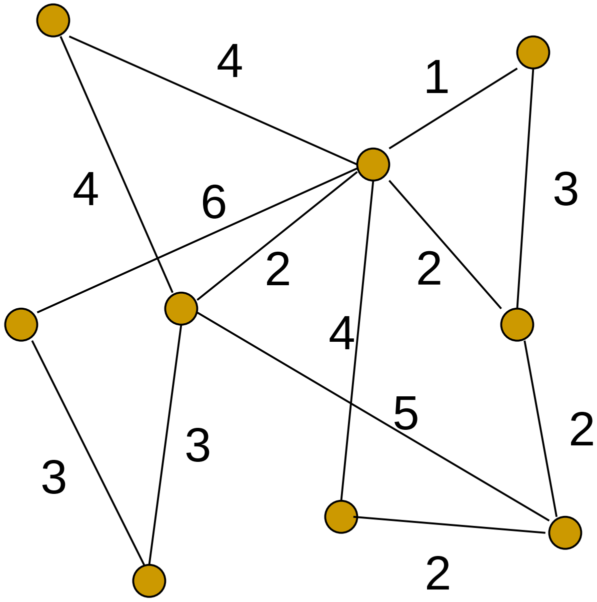
Content

- Two problems
    - Minimum Spanning Tree
    - Huffman encoding
- One approach: greedy algorithms

Example 1: Minimum Spanning Tree

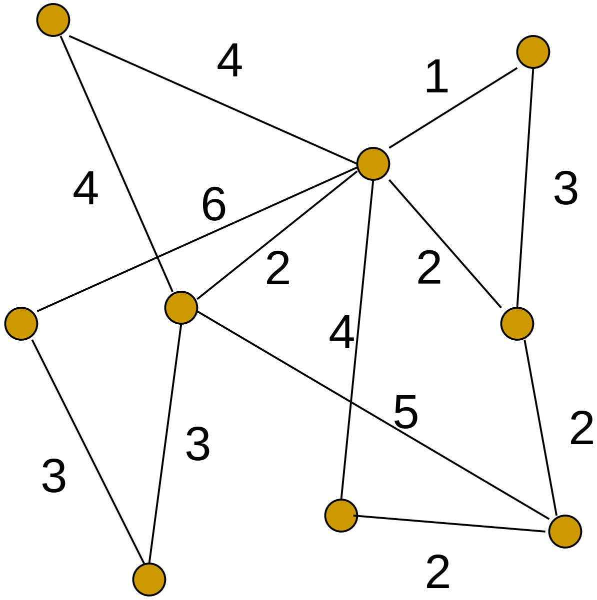
MST: Problem and Motivation

- Suppose we have n computers, connected by wires as given in the graph.
- Each wire has a renting cost.
- We want to select some wires, such that all computers are connected (i.e. every two can communicate).
- Algorithmic question: How to select a subset of wires with the minimum renting cost?
- Answer to this graph?



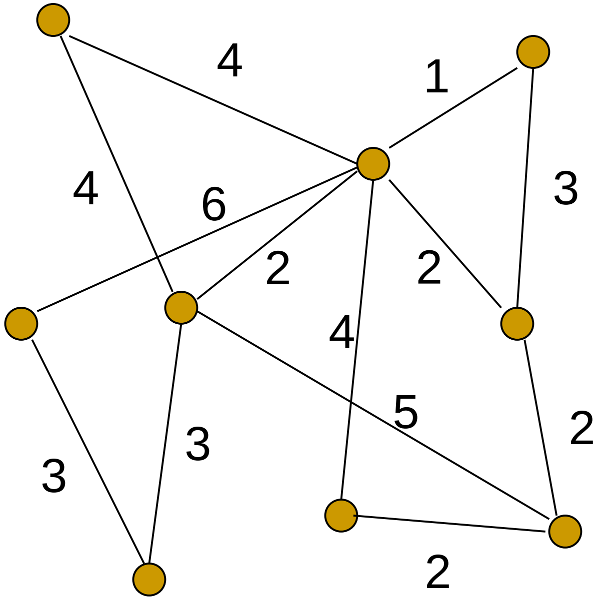
More precisely

- Given a weighted graph G, we want a subgraph $G' = (V, E'), E' \subseteq E$, s.t.
  - all vertices are connected on G'.
  - total weight $\sum_{(x,y)\in E'} w(x,y)$ is minimized.
- Observation: The answer is a tree.
  - Tree: connected graph without cycle
- *Spanning tree*: a tree containing all vertices in G.
- Question: Find a spanning tree with minimum weight.
  - The problem is thus called Minimum Spanning Tree (MST).



MST: The abstract problem

- Input: A connected weighted graph
    - $G = (V, E),\ w: E \to \mathbb{R}$.
- Output: A spanning tree with min total weight.
    - A spanning tree whose weight is the minimum of that of all spanning trees.
- Any algorithm?

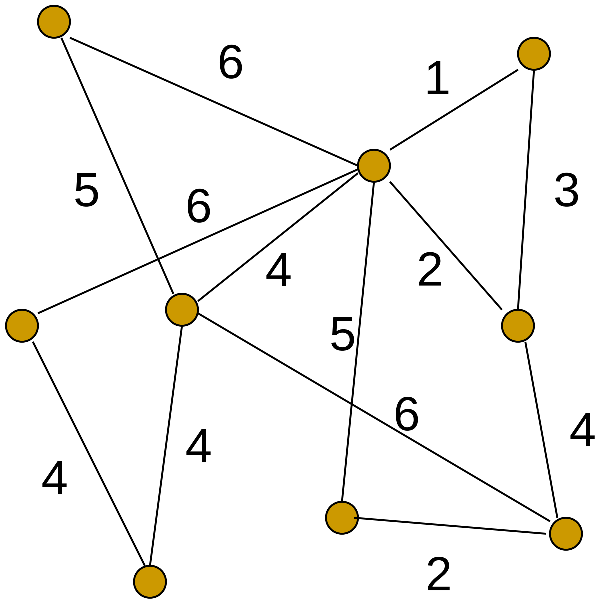


- Methodology 4: Starting from a naïve solution
  - See whether it works well enough
  - If not, try to improve it.

- A first attempt may not be correct
- But that’s fine. The key is that it’ll give you a chance to understand the problem.

What if I'm really stingy?

- I'll first pick the cheapest edge.
- I'll then again pick the cheapest one in the remaining edges
- I'll just keep doing like this …
  - as long as no cycle caused
- … until a cycle is unavoidable. Then I've got a spanning tree!
  - No cycle.
  - Connected: Otherwise I can still pick something without causing a cycle.
- Concern: Is there a better spanning tree?

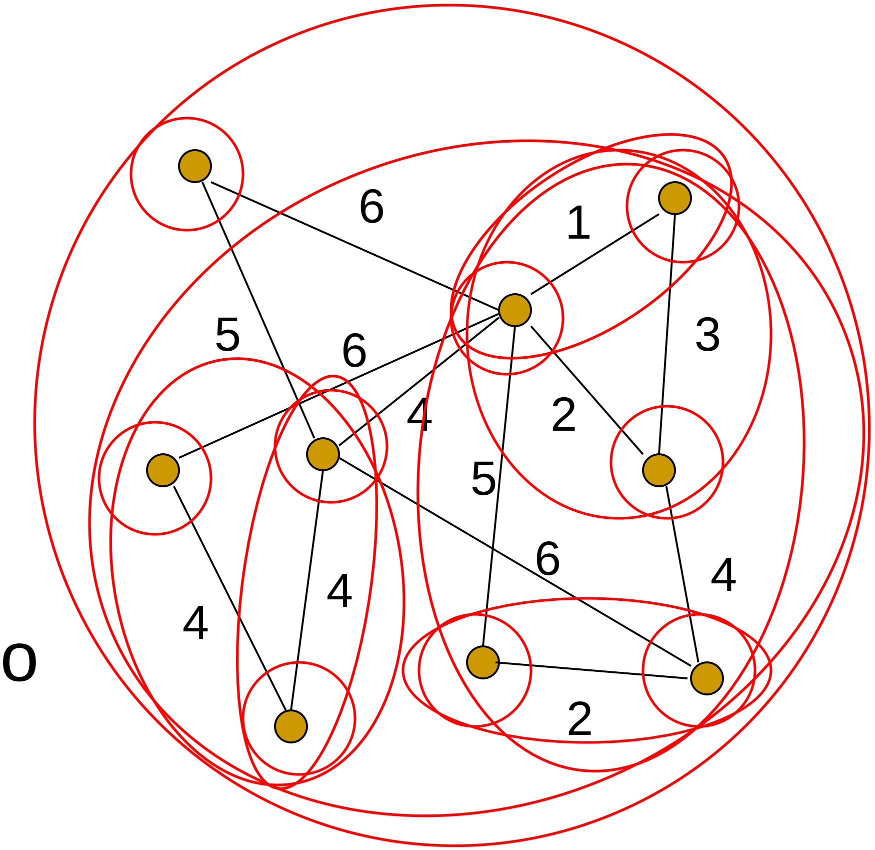


Kruskal's Algorithm

- What we did just now is Kruskal’s algorithm.
  - Repeatedly add the next lightest edge that doesn't produce a cycle…
    - in case of a tie, break it arbitrarily.
  - …until finally reaching a tree --- that’s the answer!

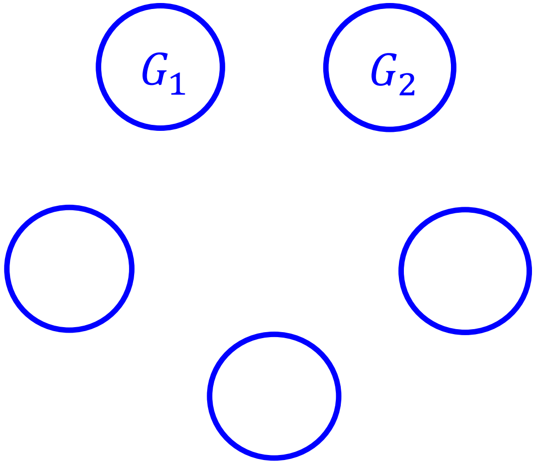
Illustrate an execution of the algorithm

- At first all vertices are all separated.
- Little by little, they merge into groups.
- Groups merge into larger groups.
- Finally, all groups merge into one.
- That's the spanning tree outputted by the algorithm.

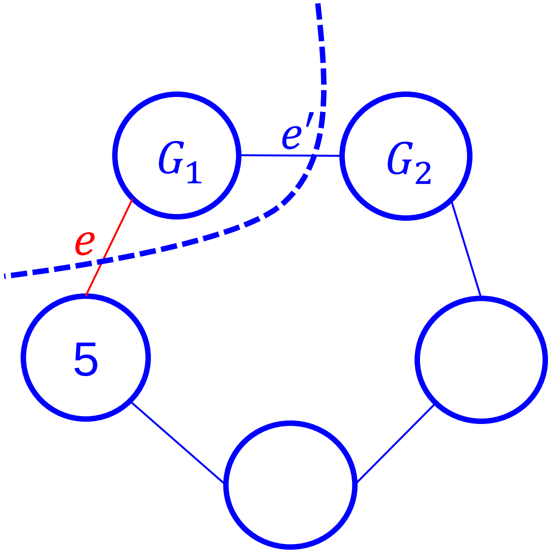


Correctness: prove by induction

- Proof plan: We will use induction to prove that at any point of time, the edges found are part of an MST.
- At any point of time, we've found some edges $M \subseteq E$,
    - M connects vertices into groups $G_1, \ldots, G_k$.
- By induction, M belongs to some MST T.



Correctness: prove by induction



- Suppose Kruskal's algorithm picks e' in the next step, connecting, say, G_1 and G_2.
- If $e' \in T$, done. If $e' \notin T$, adding e' into T would produce a cycle.
- The cycle must cross the cut $(G_1, V - G_1)$ via at least one other edge e.
- Since e' is the lightest one among all crossing edges, $w(e') \leq w(e)$.
- Let $T' = T - e + e'$, then $w(T') \leq w(T)$.
- T' is also a spanning tree.
  - Connected, and has $n - 1$ edges.
- So T' is also an MST. Induction step done.

Implementing Kruskal's Algorithm:

- Initialization:
  - Sort the edges E by weight
  - create $\{v\}$ for each $v \in V$
  - $T = \{\}$
- for all edges $(u, v) \in E$, in increasing order of weight:
  - if adding (u, v) doesn't cause a cycle
    - add edge (u, v) to T
- *Question*: What's not clearly specified yet?

Implementation

- What do we need?
- We need to maintain a collection of groups
  - Each group is a subset of vertices
  - Different subsets are disjoint.
- For a pair (u, v), we want to know whether adding this edge causes a cycle
  - If u and v are in the same subset now, then adding (u, v) will cause a cycle. Also true conversely.
  - So we need to find the two subsets containing u and v, resp.
- If no cycle is caused, then we merge the two sets containing u and v.

Data structure

- *Union-Find* data structure for disjoint sets
  - find(x): to which set does x belong?
  - union(x, y): merge the sets containing x and y.

- Using this terminology, let’s re-write the algorithm and analyze the complexity…

Kruskal's Algorithm: rewritten, complexity

- Initialization:
  - Sort the edges E by weight - $O(|E|\log|E|)$
  - create $\{v\}$ for each $v \in V$ - $O(|V|)$
  - $T = \{\}$ - $O(1)$
- for all edges $(u, v) \in E$, in increasing order of weight:

  if find$(u) \neq$ find(v) - 2*cost-of-find
  - add edge (u, v) to T - $O(1)$
  - union(u, v) - cost-of-union
- How many finds?
  - $2|E|$
- How many unions?
  - $|V| - 1$
- Total: $O(|E|\log|E| + |V| + |E|$ find-cost $+|V|$ union-cost)

data structure for union-find

- We have used various data structures: queue, stack, tree.
- Rooted Tree is good here
    - It's efficient: have/cover n leaves with only $\log_d n$ depth
        - where d is the number of children of each node.
    - Each tree has a natural id: the root
- We now use a tree for each connected component.
    - find: return the root
        - So cost-of-find depends on height(tree). Want: small height.
    - union: somehow make the two trees into one
        - The union cost … depends on implementation

union

- Recall: a tree is constructed by a sequence of union operations.
- So we want to design a union algorithm s.t.
  - the resulting tree is short
  - the cost of union itself is not large either.
- A natural idea: let the shorter tree be part of the higher tree
  - Actually right under the root of the higher tree
- To this end, we need to maintain the height information of a tree, which is pretty easy.

Details for union(x, y):

- $r_x = \text{find}(x)$
- $r_y = \text{find}(y)$
- if $height(r_x) < height(r_y)$:

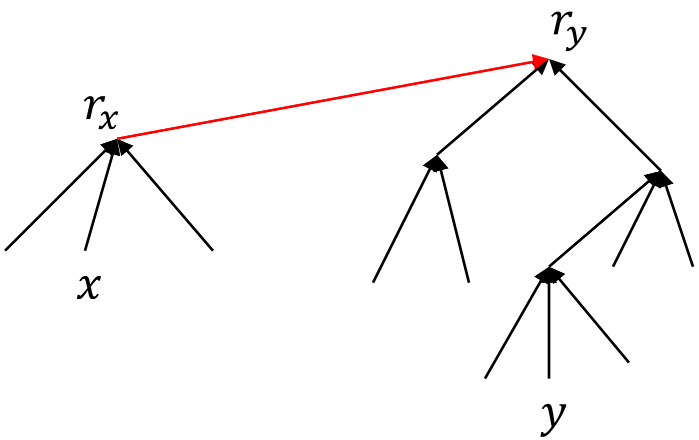
    $parent(r_x) = r_y$

- $else$

    $parent(r_y) = r_x$

    if $height(r_x) = height(r_y)$

        $height(r_y) = height(r_y) + 1$



How good is this?

- How high will the resulting tree be?
- [Claim] Any node of height h has a subtree of size at least 2^h.
  - Height of node v: height of the subtree under v. size: # of nodes
  - Proof: Induction on h.
  - The height increases (by 1) only when two trees of equal height h merge.
  - By induction, each tree has size $\geq 2^h$, now the new tree has size $\geq 2 \cdot 2^h = 2^{h+1}$. Done.
- Thus the height of a tree at any point is never more than $\log |V|$.
  - So the cost of find is at most $\log |V|$.
  - And thus the cost of union is also $O(\log|V|)$

Cost of union?

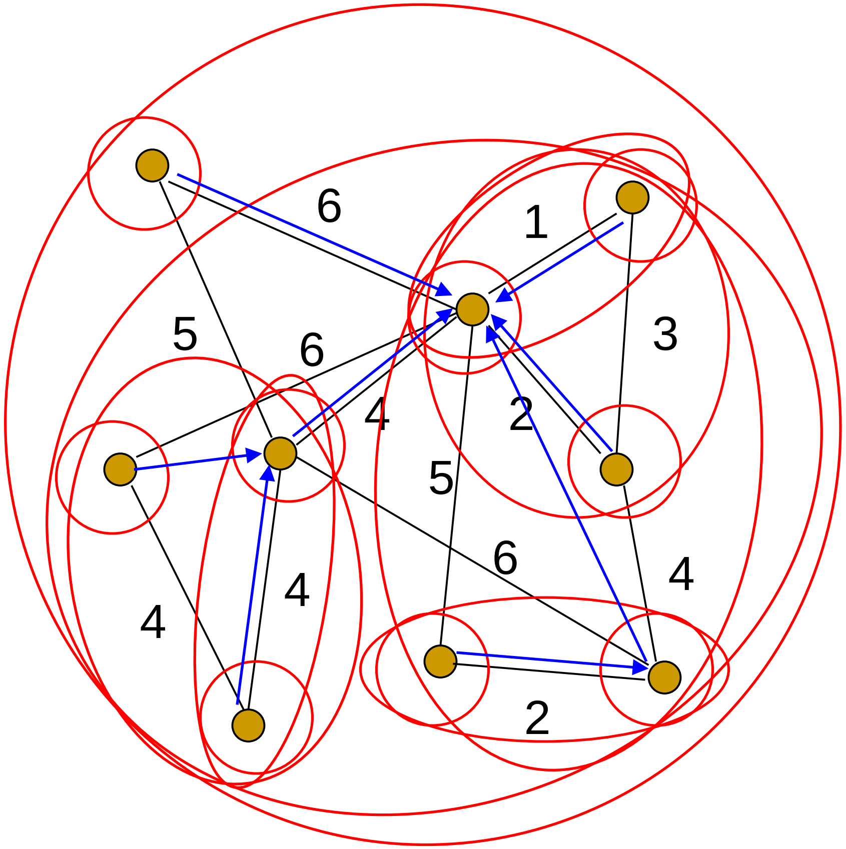
- $r_x = find(x)$ - $O(\log |V|)$
- $r_y = find(y)$ - $O(\log |V|)$
- if $height(r_x) > height(r_y)$:
    - $parent(r_y) = r_x$ - $O(1)$
- else
    - $parent(r_x) = r_y$ - $O(1)$
    - if $height(r_x) = height(r_y)$
        - $height(r_y) = height(r_y) + 1$ -$O(1)$
- Total cost of union: $O(\log |V|)$.
- Total cost of Kruskal's algorithm:

  $O(|E|\log|E| + |V| + |E|$ find-cost $+|V|$ union-cost)

  $= O(|E|\log|E| + |V| + |E|\log|V| + |V|\log|V|) = O(|E|\log|V|)$.

Don’t confuse the two types of trees

- Type 1: (parts of) the spanning tree
    - Red edges
- Type 2: the tree data structure used for implementing union-find operations
    - Blue edges



Question?

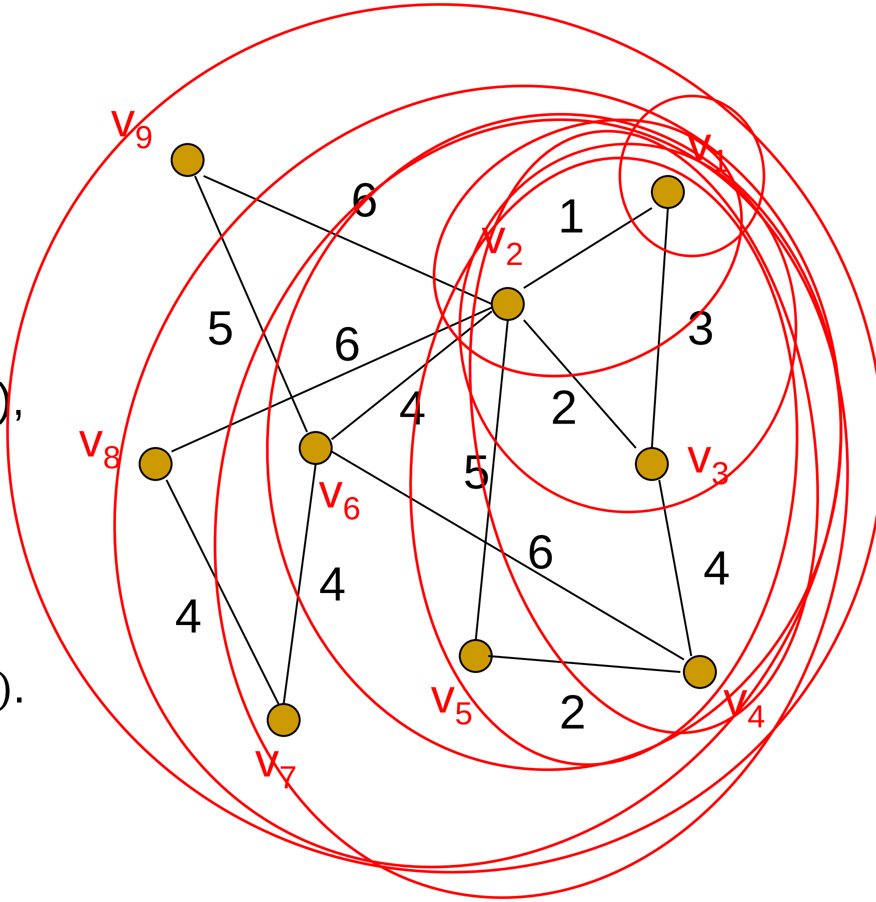
- Next: another MST algorithm.

Next: another MST algorithm

- In Kruskal’s algorithm, we get the spanning tree by merging smaller trees.
- Next, we’ll present an algorithm that always maintains one tree through the process.
- The size of the tree will grow from 1 to $|V|$.
- The whole algorithm is reminiscent of Dijkstra’s algorithm for shortest paths.

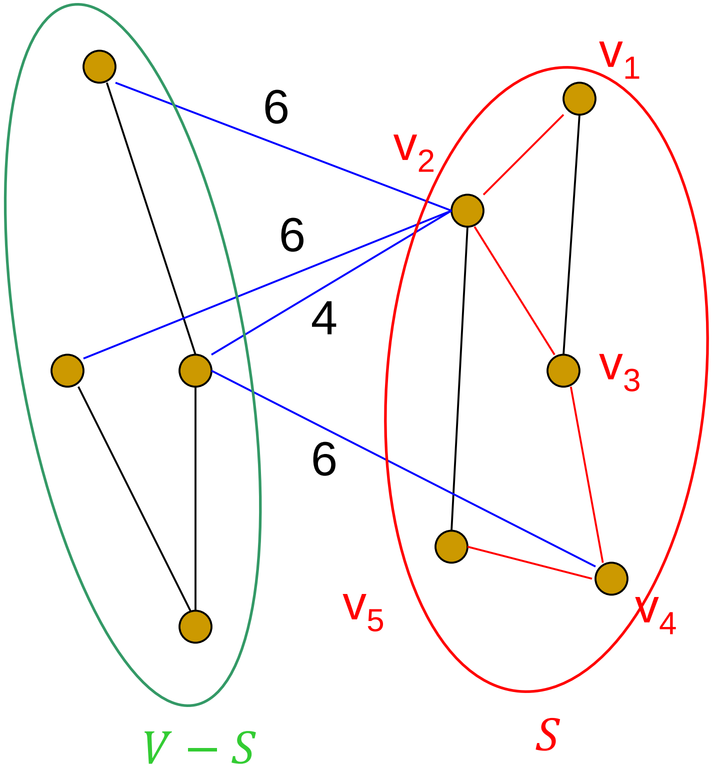
Execution on the same example

- We first pick an arbitrary vertex v_1 to start with.
  - Maintain a set $S = \{v_1\}$.
- Over all edges from v_1, find a lightest one. Say it's (v_1, v_2).
  - $S \leftarrow S \cup \{v_2\}$
- Over all edges from $\{v_1, v_2\}$ (to $V - \{v_1, v_2\}$), find a lightest one, say (v_2, v_3).
  - $S \leftarrow S \cup \{v_3\}$
- …
- In general, suppose we already have the subset $S = \{v_1, \dots, v_i\}$, then over all edges from S to $V - S$, find a lightest one (v_i, v_{i+1}).
- Update: $S \leftarrow S \cup \{v_{i+1}\}$
- …
- Finally we get a tree. That's the answer.



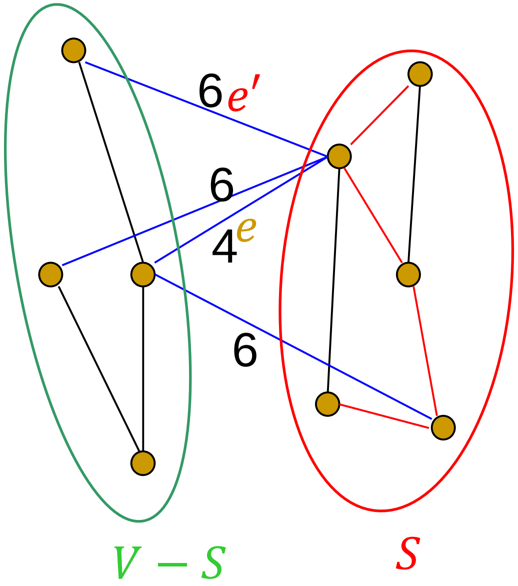
Key property

- Currently we have the set S.
- We want to main the following property:
  - The edges picked form a tree T_S in S
  - The tree T_S is part of a correct MST T.
- When adding one more node from $V - S$ to S, we want to keep the property.
- *Question: Which node to add?*
- Recall Methodology 2: Good properties often happen at extremal points.
- Finally, $S = V$, thus the property implies that our final tree is a correct MST for G.



Key property: T_S is part of a MST T.

- Consider all edges from S to $V - S$: We pick the lightest one e (and add the end point in $V - S$ to S).
- Will show: $T_S \cup \{e\}$ is part of some MST.
- By induction, ∃ a MST T containing T_S.
- If T contains e, done.
- Else: adding e into T produces a cycle.
- The cycle has some other edge(s) e' crossing S and $V - S$.
- Replacing e' with e :
  - Removing any edge in the cycle makes it still a spanner tree.
  - T is only better: $w(e) \le w(e')$

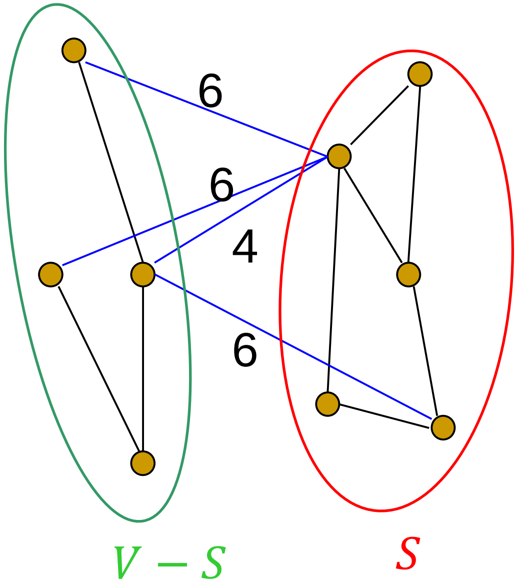


Prim's algorithm

- Implementation: Very similar to Dijkstra's algorithm.
- Now the cost function for a vertex v in $V - S$ is the minimal weight $w(v, u)$ over all $u \in S$.
  - Details omitted; see textbook.
- Complexity: also $O(|E|\log|V|)$ if we use binary min-heap as before.
  - $O(|E| + |V|\log|V|)$ if Fibonacci heap is used.

Extra: Divide and Conquer?

- Consider the following algorithm:
  - Divide the graph into two balanced parts.
    - About $n/2$ each.
  - Find a lightest crossing edge e
  - $T = T + \{e\}$
  - Recursively solve the two subgraphs.
- Is this correct?



Example 2: Huffman code

Huffman encoding

- Suppose that we have a sequence s of symbols $s_1, s_2, \dots, s_T$.
- Each s_i comes from an alphabet Γ of size n.
  - e.g. $s = (A, B, B, D, C, A, B, D)$, $\Gamma = \{A, B, C, D\}$.
- The symbols $x_1, x_2, \dots, x_n$ in Γ appear in different frequencies $f_1, f_2, \dots, f_n$.
  - f_i: the number of times x_i appears in s.
  - In earlier example: $f_1 = 2, f_2 = 3, f_3 = 1, f_4 = 2$.
- Goal: encode symbols in Γ s.t. the sequence s has the shortest length.

Example

- $\Gamma = \{A, B, C, D\}, n = 4$.
- $f_1 = 20, f_2 = 10, f_3 = 5, f_4 = 5$.
- Naive encoding:

  $$A \to 00, B \to 01, C \to 10, D \to 11.$$
- Number of bits: $(20 + 10 + 5 + 5) * 2 = 80$.
- Consider this:

  $$A \to 0, B \to 11, C \to 100, D \to 101.$$
- Number of bits:

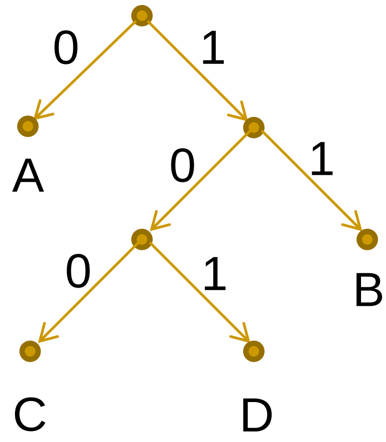
  $$20 * 1 + 10 * 2 + 5 * 3 + 5 * 3 = 70.$$

Requirement for the code

- The length can be variable: different symbols can have codeword with different lengths.
- *Prefix free*: no codeword can be a prefix of another codeword.
- Otherwise, say if the codewords are
  $$A \to 0, B \to 01, C \to 11, D \to 001$$
  then 001 is ambiguous
  - It can be either AB or D.
- *Question: How to construct an optimal prefix-free code?*

Prefix-free code and binary tree

$A \to 0, B \to 11, C \to 100, D \to 101$



- Optimal prefix-free code ↔ a full binary tree.
  - Full: each internal node has two children.
- symbol ↔ leaf.
- Encoding x_i: the path from root to the node for x_i
- Decoding:
  - Follow path to get symbol.
  - Return to the root.

Path: represented by sequence of 0's and 1's.
0: left branch. 1: right branch

Optimal tree?

- Recall question: construct an optimal code.
    - Optimal: the total length for s is minimized.
- New question: How to construct an optimal tree T.
- Namely, find $\min cost(T)$, where

$$cost(T) = \sum_{l:leaf} depth(l) \cdot f_l$$

- Recall Methodology 3: Analyze properties of an optimal solution.

In an optimal tree

- [Fact] The two symbols s_i, s_j with the smallest frequencies are at the bottom, as children of the lowest internal node.
  - Otherwise, say s_i isn't, then switch it and whoever is at the bottom. This would decrease the cost.
- This suggests a greedy algorithm:
  - Find s_i, s_j with the smallest frequencies.
  - Add a node v, as the parent of s_i, s_j.
  - Remove s_i, s_j and add v with frequency $f_i + f_j$.
  - Repeat the above until a tree with n leaves is formed.

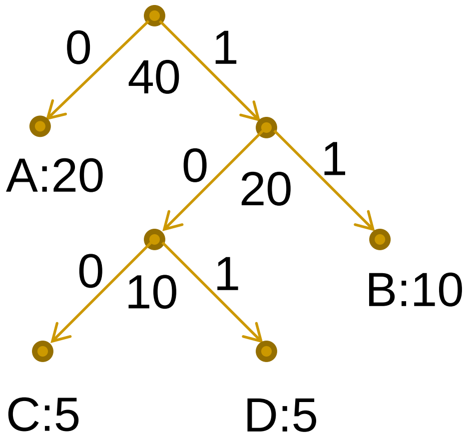
Algorithm, formal description

- Input: An array $f[1, \ldots, n]$ of frequencies
- Output: An encoding tree with n leaves

```
let $H$ be a priority queue of integers, ordered by $f$
**for** $i = 1$ to $n$
    insert$(H, i)$
**for** $k = n + 1$ to $2n - 1$
    $i =$ delete-min$(H)$; $j =$ delete-min$(H)$
    create a node numbered $k$ with children $i, j$
    $f[k] = f[i] + f[j]$
    insert$(H, k)$
```

On the running example…

- $f_1 = 20, f_2 = 10, f_3 = 5, f_4 = 5.$
- $f_1 = 20, f_2 = 10, f_5 = 5 + 5 = 10.$
- $f_1 = 20, f_6 = 10 + 10 = 20.$
- $f_7 = 20 + 20 = 40.$



- Final cost: $20 * 1 + 10 * 2 + 5 * 3 + 5 * 3 = 70$
- Also: $= \sum_{v:\text{non}-\text{root node}} \text{number for } v$
  - Including both leaves and internal nodes, but not root.

Summary

- We give two examples for greedy algorithms.
  - MST, Huffman code
- General idea: Make choice which is the best at the moment only.
  - without worrying about long-term consequences.
- An intriguing question: When greedy algorithms work?
  - Namely, when there is no need to think ahead?
- Matroid theory provides one explanation.
  - See CLRS book (Chapter 16.4) for a gentle intro.