Graph Algorithms

Acknowledgement: The notes have drawn heavily from the following books: (1) Éva Tardös and Jon Kleinberg, “Algorithm Design”, Addison-Wesley, (2) Michael T. Goodrich and Roberto Tamassia,“Algorithm Design and Applications”, Wiley, (3) Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, “Introduction to Algorithms”, The MIT Press.

1

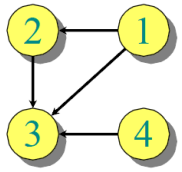
Quick Refresh of Graphs

- A Graph G = (V, E) is an ordered pair consisting of a set V of vertices (nodes) and a set of edges (links) E ⊆ V × V. An edge set E consists of pair of vertices [the pair is unordered for undirected graphs and ordered for directed graphs]. In either case, |E| O(V^2); moreover if G is connected |E| ≥ |V| – 1.
- **Adjacency matrix** of G = (V, E), where V = {1, 2, …, n} is given by

$$A[i,j] = \begin{cases} 1 \text{ if } (i,j) \in E \\ 0 \text{ if } (i,j) \notin E \end{cases} \quad \text{This takes } O(V^2) \text{ storage}$$

- **Adjacency-List representation**

An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



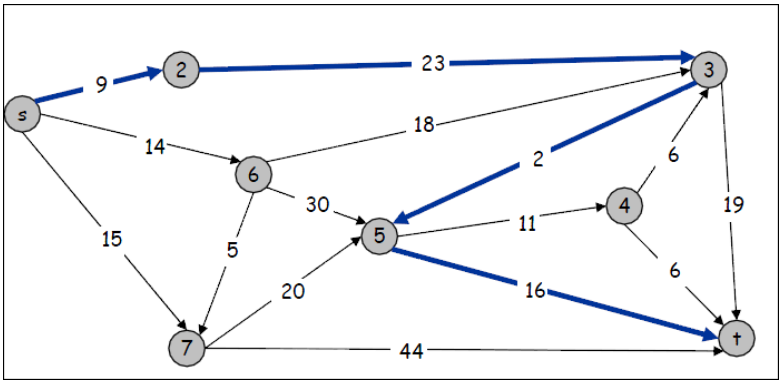
$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

For undirected graphs, $|Adj[v]| = degree(v)$.
For digraphs, $|Adj[v]| = out\text{-}degree(v)$.

- Adjacency lists need O(|V| + |E|) storage – the so called sparse representation vs. dense representation using adjacency matrix.
- When to use which one – depends on application

Shortest Path(s) in a Graph

- Assume a that each edge of a graph is associated with a weight function w: E → R^+ (only non-negative weights). For simplicity, assume that all edge weights are distinct (this is not necessary).
- Shortest Path Problem: Given a source node, say s, and a destination node, say t, find the shortest (cost of the path is the sum of the costs of all edges lying on the path) path from s to t.
- Example:

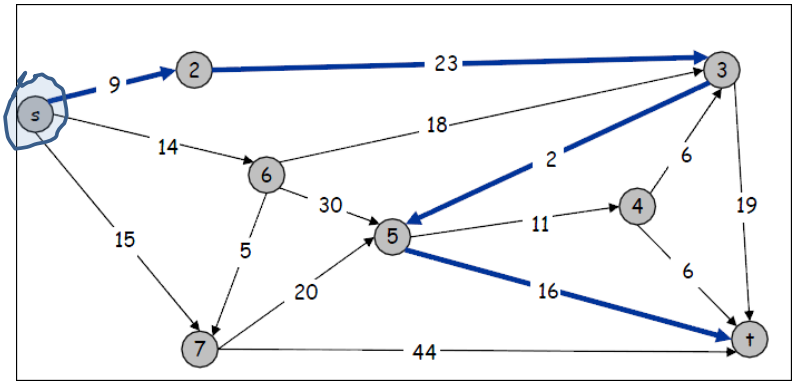


- Cost of the shortest path P from s to t = w(P) = 9 + 23 + 2 +16 = 50, where P = (s→2 →3 → 5 → t) .
- It is easy to see that there may exist more than one shortest paths.
- We are interested to find one such shortest path from s to t.

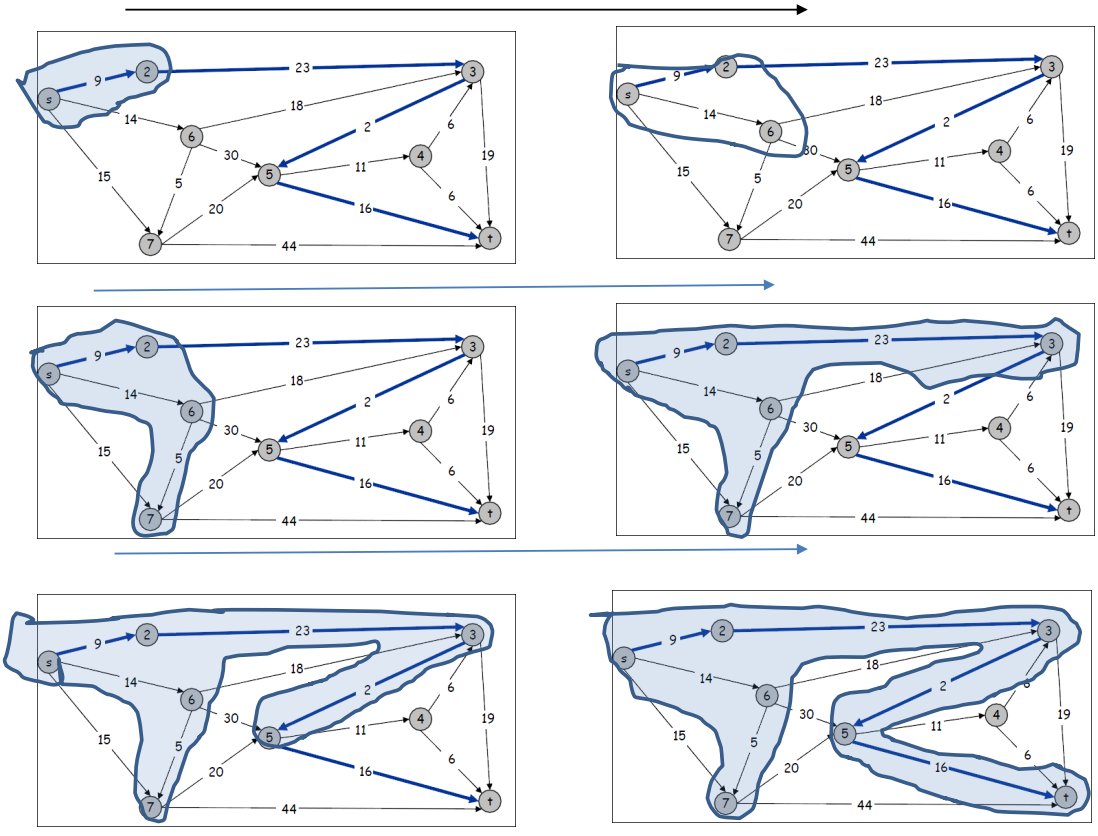
- **Approach**: We start from source s; we know the nodes directly reachable from s; none of them is the destination t in our example. Think of 2 things: (1) we have not seen t yet; the path from s to t must pass through one of the neighbors of s; (2) we sure need to explore the entire graph to compute the shortest path. [Note that the discussion applies equally well to both directed and undirected graphs with minimal changes].

Dijkstra's Algorithm

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0. [d(v) will denote the shortest path length from s to v]
- while S ≠ V
  - Select a node v ∉ S with at least one edge from *S* for which [$\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + w(e)$ is as small as possible and set S = S ∪ {v} + w(e) and d(v) = π(v).
- To produce the s-u paths corresponding to the distances found, we simply record the edge (u, v) on which it achieved the value $\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + w(e)$ [simple recursive backtracking will do]. Consider the previous graph; initially, the situation is as follows [the shaded area is S.

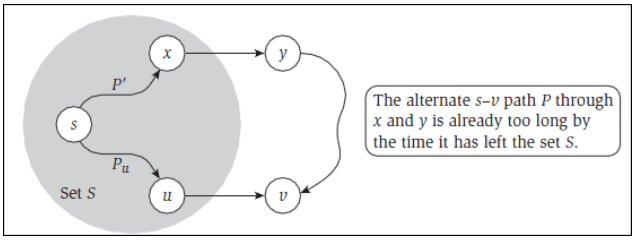


Dijkstra's Algorithm: Example Execution



Dijkstra's Algorithm: Proof of Correctness

- The Invariant is: For each node u ∈ S, d(u) is the length of the shortest s-u path [s is the given source node].
- We will prove by induction on |S|.
- Base case: Initially, S = {s}, or |S| = 1; the claim is trivially true.
- Inductive hypothesis: Assume true for |S| = k ≥ 1
  - Let v be next node added to S, and let u-v be the chosen edge.
  - The shortest s-u path plus (u, v) is an s-v path of length π(v).
  - Consider any s-v path P. We'll see that it's no shorter than π(v).
  - Let x-y be the first edge in P that leaves S, and let P' be the subpath to x; P is already too long as soon as it leaves S.



$$w(P) \geq w(P') + w(x,y) \geq d(x) + w(x,y) \geq \pi(y) \geq \pi(v).$$

non-negative wts. | ind. hypothesis | defn. of π | v is chosen over y

Dijkstra's Algorithm: Implementation

- There are n − 1 iterations of the while loop for a graph with n nodes, as each iteration adds a new node v to S.
- We explicitly maintain the values of the minima $\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + w(e)$ for each node $v \in V - S$; we maintain this information in a min-binary_heap [we have seen this simple data structure before]; select_min operation is done in O(log n) worst case time, n is the number of elements in the heap.
- Next node to explore = node with minimum $\pi(v)$.
- When exploring v, for each incident edge e = (v, w), update $\pi(w) = \min \{\pi(w), \pi(v)+w(e)\}$
- We get an implementation with O(m log n) worst case time.
- We can get a better amortized time of $O(m \log_{m/n} n)$ if we use a Fibonacci heap. **Try to prove it.**

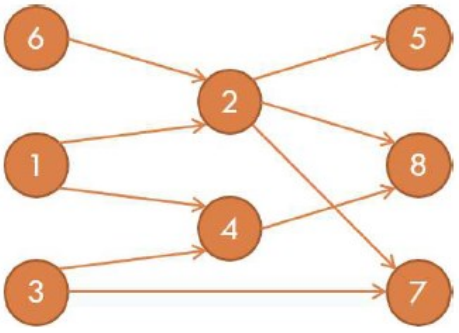
```
DIJKSTRA (V, E, s)
Create an empty priority queue.
FOR EACH v ≠ s : d(v) ← ∞; d(s) ← 0.
FOR EACH v ∈ V : insert v with key d(v) into priority queue.
WHILE (the priority queue is not empty)
    u ← delete-min from priority queue.
    FOR EACH edge (u, v) ∈ E leaving u:
        IF d(v) > d(u) + w(u, v)
        decrease-key of v to d(u) + w(u, v) in priority queue.
        d(v) ← d(u) + w(u, v).
```

Graph Traversal

- The most basic graph algorithm that visits nodes of a graph in certain order
- Used as a subroutine in many other algorithms
- Two Simple Traversals: (Assume symmetric or undirected graphs)
  - Depth-First Search (DFS): uses recursion (stack)
  - Breadth-First Search (BFS): uses queue
- **DFS(v):** visits all the nodes reachable from the given node v in depth-first order
  - Mark v as visited (how?)
  - For each edge v → u: If u is not visited, call DFS(u).
  - **Note**: non-recursive version replaces recursive calls with a stack (user maintained)
- **BFS(v):** visits all the nodes reachable from v in breadth-first order
  - Initialize a queue Q
  - Mark v as visited and push it to Q
  - While Q is not empty:
    - Take the front element of Q and call it w
    - For each edge w → u: If u is not visited, mark it as visited and push it to Q

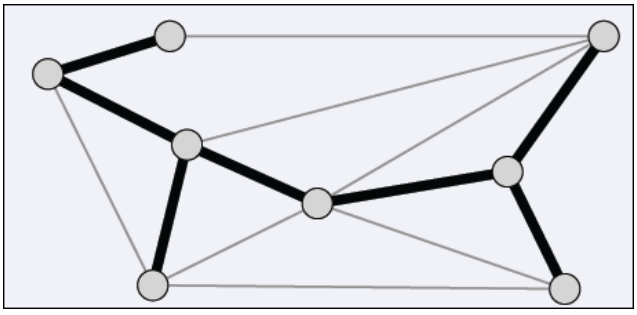
Topological Sort

- **Input**: a DAG G = (V, E)
- **Output**: an ordering of nodes such that for each edge u → v, u comes before v
- Explain Precedence Graph.
- **Note**: There can be multiple answers, e.g., both {6, 1, 3, 2, 7, 4, 5, 8} and {1, 6, 2, 3, 4, 5, 7, 8} (and others) are valid orderings for the graph below



Spanning Tree Properties

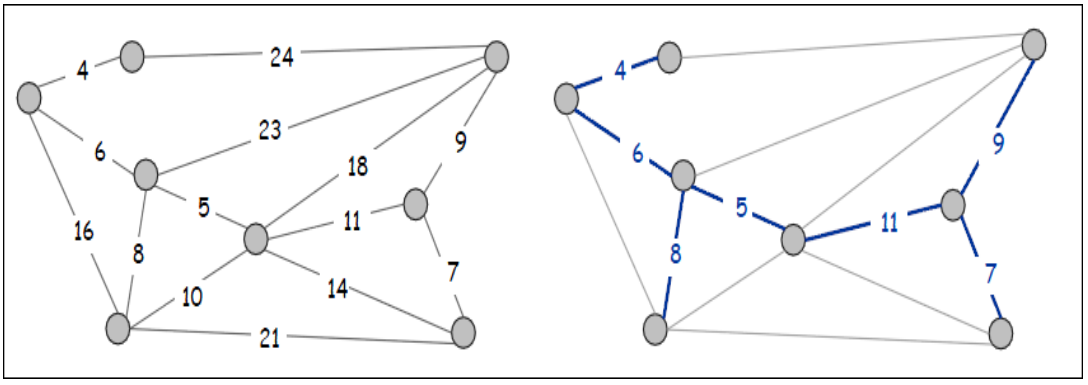
- Consider a **connected, undirected** graph G = (V, E); Let T = (V, F), F $\subseteq$ E, be a subgraph of G = (V, E). All of the following statements must be true:
  - T is a spanning tree of G.
  - T is acyclic and connected.
  - T is connected and has n – 1 edges.
  - T is acyclic and has n – 1 edges.
  - T is minimally connected: removal of any edge disconnects it.
  - T is maximally acyclic: addition of any edge creates a cycle.
  - T has a unique simple path between every pair of nodes.



The black edges constitute the spanning tree. Satisfy yourself that all those statements are true [some are equivalent]

Minimal Spanning Tree

- A connected, undirected graph G = (V, E) with weight function w: E→ $\Re^+$ (set of +ve reals); we'll assume all edge weights are distinct [Explain briefly the general case when duplicates are allowed]
- A connected acyclic subgraph T of G that spans all vertices is called a spanning tree of the graph. A minimal spanning tree (MST) is a spanning tree of minimum weight, i.e., a spanning tree whose sum of edge weights is minimized.
- Example:



- **Note:** Cayley's theorem. There are n^{n-2} spanning trees of K_n ; thus, not possible to find MST by exhaustive enumeration.

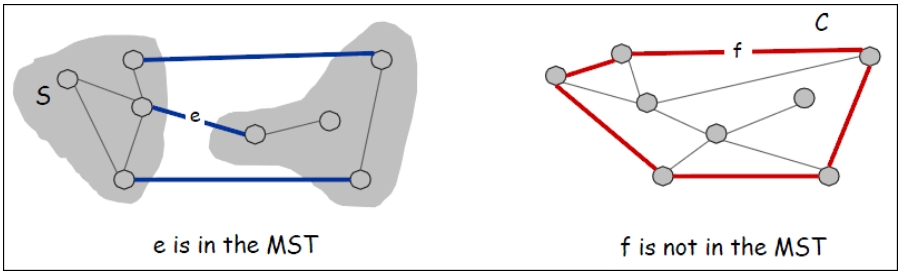
Applications

- MST is fundamental problem with diverse applications.
- Network design.
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems.
  - traveling salesperson problem, Steiner tree
- Indirect applications.
  - max bottleneck paths
  - LDPC codes for error correction
  - image registration with Renyi entropy
  - learning salient features for real-time face verification
  - reducing data storage in sequencing amino acids in a protein
  - model locality of particle interactions in turbulent fluid flows
  - autoconfig protocol for Ethernet bridging to avoid cycles in a network
- Cluster analysis.

Greedy Algorithms for MST

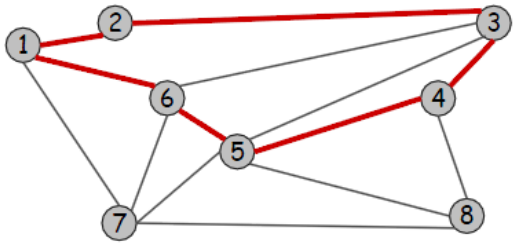
1. **Kruskal's algorithm**. Start with T = ϕ. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.
2. ~~**Reverse-Delete algorithm**. Start with T = E. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T.~~
3. **Prim's algorithm**. Start with some root node s and greedily grow a tree T from s outward. At each step, add the least weight edge e to T that has exactly one endpoint in T.

- We'll do 1 and 3. The greedy structure is the same; implementations are different, one uses the min-heaps and the other uses Union-Find data structure. Both works in O(m log n) time.
- **Cut property**. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e [cheapest way to connect S and V – S].
- **Cycle property**. Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f. [Why? Assume f belongs to an MST T1 ⇒ deleting f breaks T1 in two subtrees with the two ends of f in different subtrees ⇒ remainder of C reconnects the subtrees, hence there is an edge e of C with ends in different subtrees, i.e., it reconnects the subtrees into a tree T2 with weight less than that of T1, because the weight of e is less than the weight of f.]



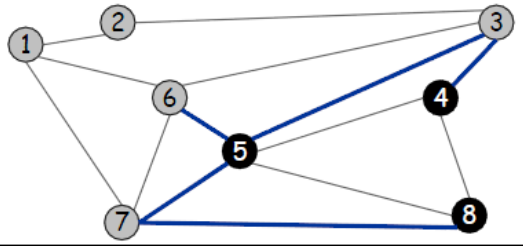
Cycles and Cuts

Cycle. Set of edges the form a-b, b-c, c-d, …, y-z, z-a.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1

Cutset. A cut is a subset of nodes S. The corresponding cutset D is the subset of edges with exactly one endpoint in S.

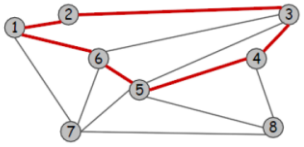


Cut S = { 4, 5, 8 }
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8

A cut is a partition of the nodes into two nonempty subsets S and V – S.

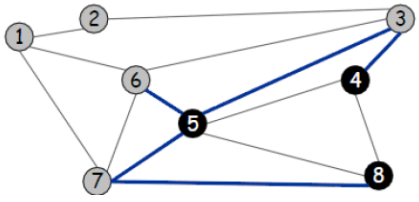
Cycles and Cuts

- Cycle: Consider an undirected graph G. A cycle is a simple (no node is visited more than once) path (sequence of edges) from any node back to itself.



Examples: (1) (1, 2, 3, 4, 5, 6, 1), (2) (1, 6, 7, 1), (3) (3, 4, 5, 6, 3), so on.

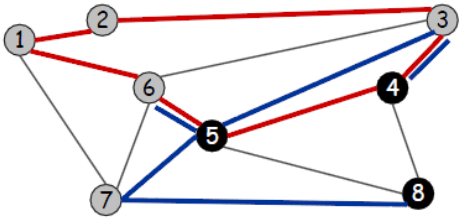
- A cut C = (S, T) is a partition of V of a graph G=(V, E) into two subsets S and T. The cut-set of a cut C=(S,T) is the set $\{(u,v) \in E \mid u \in S, v \in T\}$ of edges that have one endpoint in S and the other endpoint in T.



Example: Cut S = {4, 5, 8}, T = V – S ={1, 2, 3, 6, 7}.
Cut-set = {5-6, 5-7, 3-4, 3-5, 7-8}

Cycle-Cut Intersection

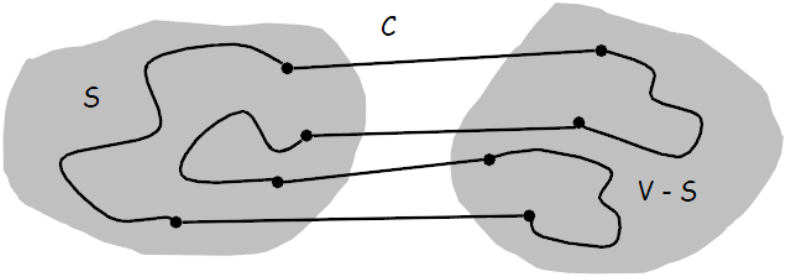
Claim. A cycle and a cutset intersect in an even number of edges.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersection = 3-4, 5-6

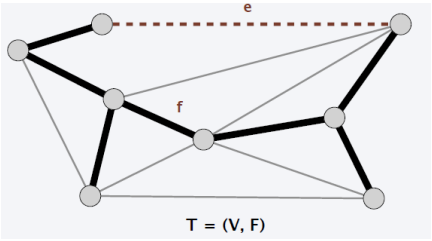
y

Pf. (by picture)



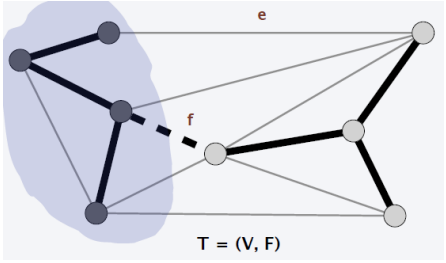
Fundamental Cycle and Fundamental Cutset

- Fundamental cycle.
  - Adding any non-tree edge e to a spanning tree T forms unique cycle C.
  - Deleting any edge f ∈ C from T ∪ { e } results in new spanning tree.



Observe: If w(e) < w(f), then T is not an MST

- Fundamental cutset.
  - Deleting any tree edge f from a spanning tree T divide nodes into two connected components. Let D be cutset.
  - Adding any edge e ∈ D to T – { f } results in new spanning tree.



Observe: If w(e) < w(f), then T is not an MST

A Simple Greedy Algorithm

- There are two simple rules: Apply the red and blue rules (in any sequence) until n – 1 edges are colored blue. The blue edges form an MST.
  - Red Rule: Select a cycle C in G with no red edges; Select an uncolored edge of C of max weight and color it red.
  - Blue Rule: Select a cut-set D with no blue edges; Select an uncolored edge in D of min weight and color it blue.

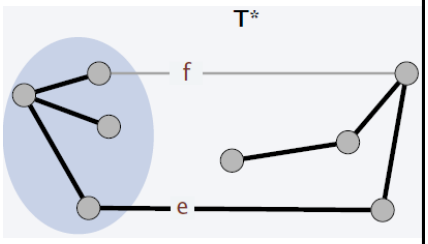
- **Color invariant.** There exists an MST T* containing all the blue edges and none of the red edges.
- We will prove by induction on the number of iterations;
  - **Base case**. No edges colored ⇒ every MST satisfies invariant.
  - We consider the two rules separately to show that the invariant is true.

Proof of Correctness

- **Induction step (red rule).** Suppose color invariant true before red rule.
  - let C be chosen cycle, and let e be edge colored red.
  - if $e \notin T^*$, T^* still satisfies invariant.
  - Otherwise, consider fundamental cut-set D by deleting e from T^*.
  - let $f \in D$ be another edge in C.
  - f is uncolored and $w(e) \geq w(f)$ since (1) $f \notin T^* \Rightarrow$ f not blue, (2) red rule $\Rightarrow$ f not red and $w(e) \geq w(f)$
  - Thus, $T^* \cup \{ f \} - \{ e \}$ satisfies invariant.

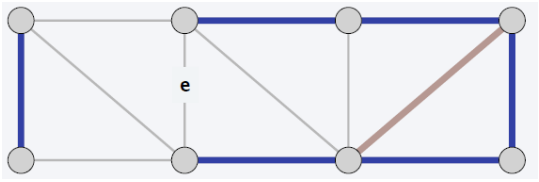
Induction step (blue rule). Suppose color invariant true before blue rule.

- let D be chosen cutset, and let f be edge colored blue.
- if $f \in T^*$, T^* still satisfies invariant.
- Otherwise, consider fundamental cycle C by adding f to T^*
- let $e \in C$ be another edge in D.
- e is uncolored and $w(e) \geq w(f)$ since (1) $e \in T^* \Rightarrow$ e not red (2) blue rule $\Rightarrow$ e not blue and $c(e) \geq c(f)$
- Thus, $T^* \cup \{ f \} - \{ e \}$ satisfies invariant.

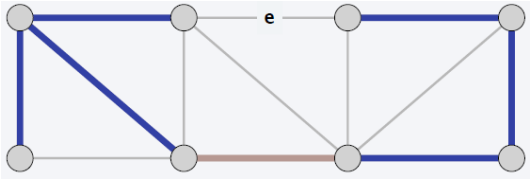


Proof of Termination

- **Theorem**. The greedy algorithm terminates. Blue edges form an MST.
- **Proof**. We need to show that either the red or blue rule (or both) applies.
  - Suppose edge e is left uncolored.
  - Blue edges form a forest.
  - Case 1: both endpoints of e are in same blue tree.
    ⇒ apply red rule to cycle formed by adding e to blue forest.
  - Case 2: both endpoints of e are in different blue trees.
    ⇒ apply blue rule to cutset induced by either of two blue trees.



Case 1



Case 2

Prim's Algorithm

- **Algorithm**: Initialize S = any node, Sort the edges in O(m log n) time, n = no. of nodes and m = no. of edges. Then repeat n – 1 times {Add to tree the min weight edge with one endpoint in S; Add new node to S.}
- **Theorem**. Prim's algorithm computes the MST. **Proof**: Special case of greedy algorithm (blue rule repeatedly applied to S).
- Implement: [d(v) = weight of cheapest known edge between v and S] The data structure and approach are almost identical to Dijkstra's algorithm for shortest path.

PRIM (V, E, w)

Create an empty priority queue.

s ← any node in V.

FOR EACH v ≠ s : d(v) ← ∞; d(s) ← 0. //Initialization

FOR EACH v : insert v with key d(v) into priority queue.

REPEAT n – 1 times

    u ← delete-min from priority queue.

    FOR EACH edge (u, v) ∈ E incident to u:

        IF d(v) > w(u, v)

            decrease-key of v to w(u, v) in priority queue.

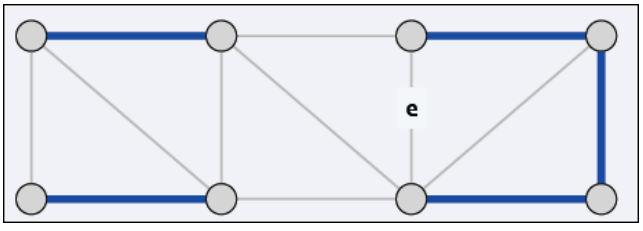
            d(v) ← w(u, v).

Kruskal's algorithm

- Consider edges in ascending order of weight:
- **Algorithm**: Add to tree unless it would create a cycle.
- **Theorem**. Kruskal's algorithm computes the MST.
- **Proof**. Special case of greedy algorithm.
  - Case 1: both endpoints of e in same blue tree.
    ⇒ color red by applying red rule to unique cycle.
  - Case 2. If both endpoints of e are in different blue trees.
    ⇒ color blue by applying blue rule to cut-set defined by either tree.

all other edges in cycle are blue

no edge in cut-set has smaller weight
(since Kruskal chose it first)



Kruskal's Algorithm: implementation

- Theorem. Kruskal's algorithm can be implemented in O(m log m) time.
- Algorithm:
  - Sort edges by weight.
  - Use union-find data structure to dynamically maintain connected components.

KRUSKAL (V, E, w)
SORT m edges by weight so that $w(e_1) \le w(e_2) \le \ldots \le w(e_m)$
$S \leftarrow \varphi$
FOREACH $v \in V$: MAKESET(v).
FOR i = 1 TO m
  (u, v) $\leftarrow e_i$
  IF FINDSET(u) $\neq$ FINDSET(v)
    $S \leftarrow S \cup \{ e_i \}$
    UNION(u, v).
RETURN S

Are u and v in the same component?

make u and v in same component

Huffman Codes & Data Compression

- The problem is how to encode symbols using bits. Consider a small alphabet: 26 English letters, the space and 5 punctuation symbols: period, question mark, exclamation point, and apostrophe. We have 32 symbols and we must have minimum 5 bits to encode each symbol (mapping of bit strings to symbols is arbitrary). We need to spend an average of 5 bits per symbol. Is it necessary?
- Note the letters in most human alphabets do not get used equally frequently. In English, for example, the letters e, t, a, o, i, and n get used much more frequently than q, j, x, and z (by more than an order of magnitude) – we will try to use *a small number of bits for the frequent letters, and a larger number of bits for the less frequent ones (***VLC – variable length codes***)*; the goal is to reduce the average number of bits per letter when we average over a long string of typical text – a fundamental problem in the area of data compression.
- We still need to solve the problem of ambiguity (there will exist pairs of letters where the bit string that encodes one letter is a *prefix* of the bit string that encodes another). We need to map letters to bit strings in such a way that no encoding is a prefix of any other (**prefix code**).
- Example: a → 0, b → 100, c → 101, d → 11. Observe, this is a valid prefix code on the alphabet {a, b, c, d}.
- Coding: aabddcaa → 0010011111010 0 [14 bits] **Prefix code ensures unique decodability.**

Optimal Prefix Code

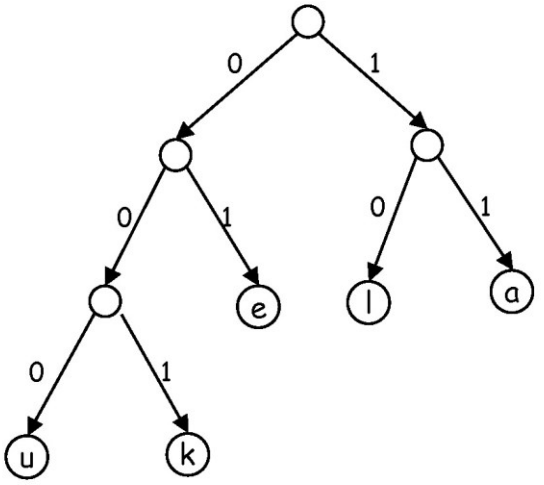
- Prefix property ensures quick and unique decode-ability. We need some notation to express the frequencies of letters to consider optimality.
- Consider a text S over an alphabet of size n; let f_x be the frequency of symbol $x \in S$; note that $\sum_{x \in S} f_x = 1$; total encoding length is $n \times \sum_{x \in S} f_x |\sigma(x)|$, where $|\sigma(x)|$ is the length of the bit string $\sigma(x)$ used to encode x. Thus, we get **average bit length (of the mapping function σ_1) is $ABL(\sigma_1) = \sum_{x \in S} f_x |\sigma_1(x)|$.**
- **Example**: Assume text with the letters S = {a, b, c, d, e}, and their frequencies are as follows: $f_a = .32$, $f_b = .25$, $f_c = .20$, $f_d = .18$, $f_e = .05$ where σ_1 [a, b, c, d, e] = [11, 01, 001, 10, 000]. We get $ABL(\sigma_1) = 2.25$; better than 3, savings of 25 percent. Is σ_1 optimal? **No**.
  - Consider another mapping σ_2 [a, b, c, d, e] = [11, 10, 01, 001, 000] for the same problem. We get $ABL(\sigma_2) = 2.23 < ABL(\sigma_1)$.

  So, the problem is to minimize ABL(σ) over all possible mapping σ, given an alphabet and a set of frequencies for the letters.

- We want to design a greedy algorithm to generate such an optimal mapping for a given alphabet and their frequencies. Note that if the frequencies change, the generated mapping may not be optimal any more.
- We need to represent Prefix codes using binary trees.

Representing Prefix Codes using Binary Trees

Ex. c(a) = 11
c(e) = 01
c(k) = 001
c(l) = 10
c(u) = 000



Q. How does the tree of a prefix code look?
A. Only the leaves have a label.
Pf. An encoding of x is a prefix of an encoding of y if and only if the path of x is a prefix of the path of y.

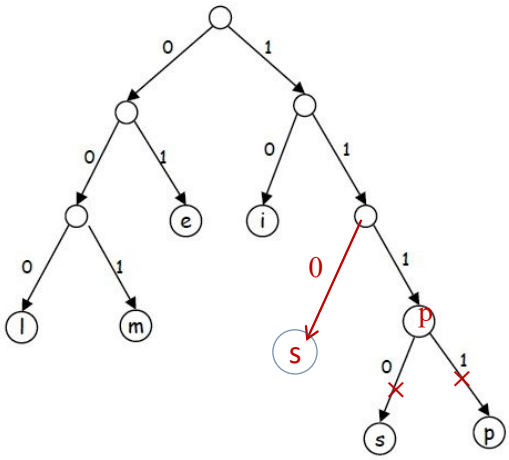
Representing Prefix Codes using Binary Trees

Q. What is the meaning of 1110100011111010 00 ?
A 'Simpel'

Q. How can this prefix code be made more efficient?
A. Change encoding of p and s to a shorter one.
This tree is now full.

Note: Since the symbols are always at the leaves, the codes are uniquely decodable and the length of the encoding of any symbol is given by the depth of x in the tree T. So we can also write

$$ABL(T) = \sum_{x \in S} f_x \cdot depth_T(x)$$



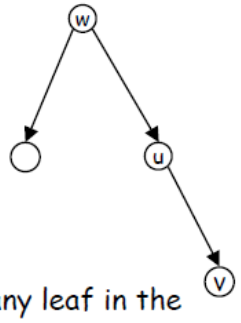
Representing Prefix Codes using Binary Trees

Definition. A tree is full if every node that is not a leaf has two children.

Claim. The binary tree corresponding to the optimal prefix code is full.

Pf. (by contradiction)

- Suppose T is binary tree of optimal prefix code and is not full.
- This means there is a node u with only one child v.
- Case 1: u is the root; delete u and use v as the root



- Case 2: u is not the root
  - let w be the parent of u
  - delete u and make v be a child of w in place of u
- In both cases the number of bits needed to encode any leaf in the subtree of v is decreased. The rest of the tree is not affected.
- Clearly this new tree T' has a smaller ABL than T. Contradiction.

Optimal Prefix Codes: Huffman Encoding

Observation. Lowest frequency items should be at the lowest level in tree of optimal prefix code.

Observation. For n > 1, the lowest level always contains at least two leaves.

Observation. The order in which items appear in a level does not matter.

Claim. There is an optimal prefix code with tree T* where the two lowest-frequency letters are assigned to leaves that are siblings in T*.

Greedy template. [Huffman, 1952] Create tree bottom-up.
Make two leaves for two lowest-frequency letters y and z.
Recursively build tree for the rest using a meta-letter for yz.



Optimal Prefix Codes: Huffman Encoding

```
Huffman(S) {
   if |S|=2 {
      return tree with root and 2 leaves
   } else {
      let y and z be lowest-frequency letters in S
      S′ = S
      remove y and z from S′
      insert new letter ω in S′ with f_ω=f_y+f_z
      T′ = Huffman(S′)
      T = add two children y and z to leaf ω from T′
      return T
   }
}
```

Q. What is the time complexity?

A. $T(n) = T(n-1) + O(n)$
so $O(n^2)$

Q. How to implement finding lowest-frequency letters efficiently?

A. Use priority queue for S: $T(n) = T(n-1) + O(\log n)$ so $O(n \log n)$

Huffman Encoding: Greedy Analysis

Claim. Huffman code for S achieves the minimum ABL of any prefix code.

Pf. by induction, based on optimality of T' (y and z removed, ω added) (see next page)

Claim. ABL(T')=ABL(T)-f_ω

Pf.

$$\begin{aligned}
\mathrm{ABL}(T) &= \sum_{x \in S} f_x \cdot \mathrm{depth}_T(x) \\
&= f_y \cdot \mathrm{depth}_T(y) + f_z \cdot \mathrm{depth}_T(z) + \sum_{x \in S, x \neq y, z} f_x \cdot \mathrm{depth}_T(x) \\
&= (f_y + f_z) \cdot (1 + \mathrm{depth}_T(\omega)) + \sum_{x \in S, x \neq y, z} f_x \cdot \mathrm{depth}_T(x) \\
&= f_\omega \cdot (1 + \mathrm{depth}_T(\omega)) + \sum_{x \in S, x \neq y, z} f_x \cdot \mathrm{depth}_T(x) \\
&= f_\omega + \sum_{x \in S'} f_x \cdot \mathrm{depth}_{T'}(x) \\
&= f_\omega + \mathrm{ABL}(T')
\end{aligned}$$

Huffman Encoding: Greedy Analysis

Claim. Huffman code for S achieves the minimum ABL of any prefix code.

Pf. (by induction)

Base: For n=2 there is no shorter code than root and two leaves.

Hypothesis: Suppose Huffman tree T' for S' with ω instead of y and z is optimal. (IH)

Step: (by contradiction)

- Suppose Huffman tree T for S is not optimal.
- So there is some tree Z such that ABL(Z) < ABL(T).
- Then there is also a tree Z for which leaves y and z exist that are siblings and have the lowest frequency (see observation).
- Let Z' be Z with y and z deleted, and their former parent labeled ω.
- Similar T' is derived from S' in our algorithm.
- We know that $ABL(Z')=ABL(Z)-f_{\omega}$, as well as $ABL(T')=ABL(T)-f_{\omega}$.
- But also ABL(Z) < ABL(T), so ABL(Z') < ABL(T').
- Contradiction with IH.

Test

- Write Something; Pradip Srimani MWrite Something; Pradip Srimani Monai, Rono