Greedy Algorithms

Stefan D. Bruda

CS 317, Fall 2024

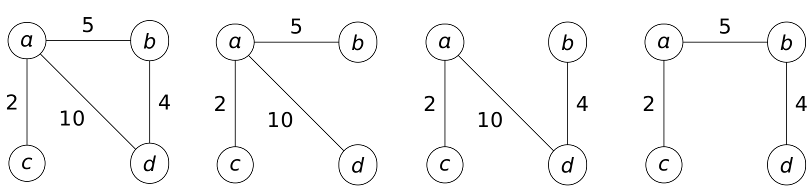
The greedy technique

- Typically suitable to for optimization problems
- Builds the solution iteratively
- Makes a locally optimum choice in ech iteration in the hope that all local optima will lead to a global optimum
- Guaranteed to give a “good” solution, but does not guarantee an optimal solution for all optimization problems

algorithm GREEDY(A: set of candidates)**:**
  $solution \leftarrow \emptyset$
  while *solution* not complete **do**
    $x \leftarrow$ SELECTBEST(A) (local optimum)
    $A \leftarrow A \setminus x$
    if FEASIBLE($solution \cup x$) **then**
      $solution \leftarrow solution \cup x$

Minimum-cost spanning trees

- A spanning tree of a graph G is a connected acyclic subgraph of G that contains all the vertices



Graph | BFS tree Cost = 17 | DFS tree Cost = 16 | Min cost spanning tree Cost = 11

- Problem: Given a weighted undirected connected graph G
- Question: Find a spanning tree of G with minimum cost
  - Many applications including transportation networks, computer networks, electrical grids, even financial markets

Kruskal’s algorithm

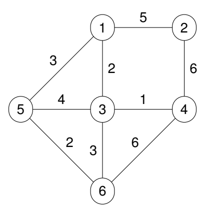
- For a given weighted graph $G = (V, E, w)$:
  - Choose an edge e of minimum weight $w(e)$
  - If the edge does not create a cycle add it to the tree

algorithm KRUSKAL($G = (V, E, w)$)**:**
  $T \leftarrow \emptyset$
  $c \leftarrow 0$
  $L \leftarrow E$
  while $|T| \leq n - 1$ **do**
    Select $e \in L$, $w(e) = \min\{w(x) : x \in R\}$
    $L \leftarrow L \setminus \{e\}$
    if $T \cup e$ does not contain cycles **then**
      $T \leftarrow T \cup \{e\}$
      $c \leftarrow c + w(e)$

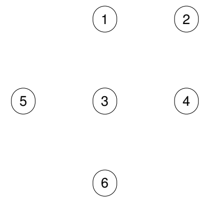
- Still to implement:
  - Find an edge with a minimum weight
  - Detect cycles
- Data structures needed:
  - List of edges sorted by weight
  - Disjoint sets representing each connected component

Kruskal's algorithm example

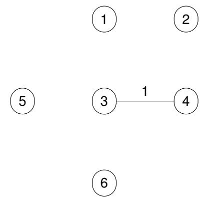
Graph:



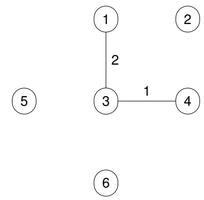
Start: six singletons



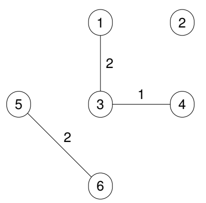
#1: choose (3,4)



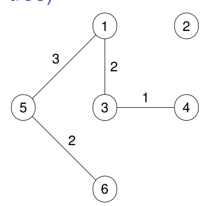
#2: choose (1,3) (or (5,6))



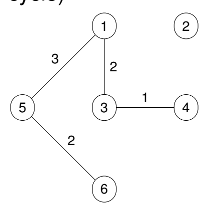
#3: choose (5,6) (or (1,3))



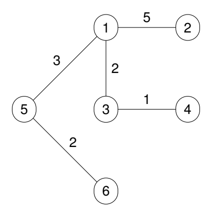
#4: choose (1,5) (or (3,6), for a different tree)



#5: choose and ignore (3,6) (creates cycle)



#6: choose (1,2) and done



Kruskal's algorithm (cont'd)

algorithm KRUSKAL($G = (V, E, w)$):
- $T \leftarrow \emptyset$
- $c \leftarrow 0$
- $L \leftarrow$ MAKEQUEUE(E)
- **for** $i = 1$ **to** n **do** MAKESET(i)
- $i \leftarrow 1$
- **while** $i \leq n - 1$ **do**
  - $(u, v) \leftarrow$ DEQUEUE(L)
  - $s_1 \leftarrow$ FINDSET(u)
  - $s_2 \leftarrow$ FINDSET(v)
  - **if** $s_1 \neq s_2$ **then**
    - UNION(s_1, s_2)
    - $T \leftarrow T \cup \{(u, v)\}$
    - $c \leftarrow c + w((u, v))$
    - $i \leftarrow i + 1$

- Choice of implementation for the priority queue:
  - Sorted list: $O(n \log n)$ to create, $O(1)$ to extract minimum
  - Min heap: $O(n)$ to create, $O(\log n)$ to extract minimum
- Running time ($|V| = n$, $|E| = m$):
  - With sorted list: $T(n) = m \log m + n + m(1 + 2 \log n) = O(m \log n)$
  - With heap: $T(n) = m + n + m(\log m + 2 \log n) = O(m \log n)$
- Correctness:
  - Loop invariant: The graph induced by each disjoint set S in (S, T) is a minimum-cost spanning tree for (S, E)
  - Kruskal's algorithm maintain a forest of minimum-cost spanning trees, collapsing it progressively into a single overall minimum-cost spanning tree

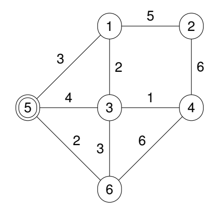
Prim's algorithm

- Maintains a single, partial minimum-cost spanning tree
  - Start with a single vertex and no edges
  - Expand the tree by greedily choosing the minimum weight edge with an end in the tree and the other end outside the tree

algorithm PRIM($G = (V, E, w), v_0 \in V$):
- $T \leftarrow \emptyset$
- $c \leftarrow 0$
- $S \leftarrow \{v_0\}$
- **while** $S \neq V$ **do**
  - Select $v \in V \setminus S$ nearest to S
  - Let $u \in S$ be the nearest vertex to v
  - $S \leftarrow S \cup \{v\}$
  - $T \leftarrow T \cup \{(v, u)\}$
  - $c \leftarrow c + w((u, v))$

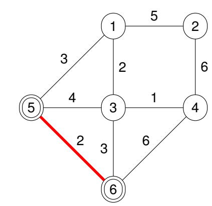
- To keep track of candidate edges for each vertex outside the tree we keep track of:
  - Its minimum distance from the tree
  - The edge that realizes that minimum distance

Prim's algorithm example



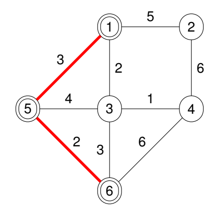
	dist	near
1	3	5
2	∞	5
3	4	5
4	∞	5
$\cancel{5}$	–	5
6	**2**	**5**

$\Rightarrow$



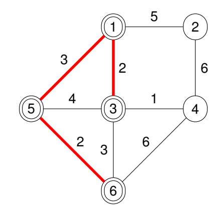
	dist	near
1	**3**	**5**
2	∞	5
3	3	6
4	3	6
$\cancel{5}$	–	5
$\cancel{6}$	2	5

$\Rightarrow$



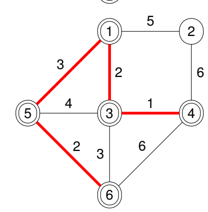
	dist	near
$\cancel{1}$	3	5
2	5	1
3	**2**	**1**
4	3	6
$\cancel{5}$	–	5
$\cancel{6}$	2	5

$\Rightarrow$



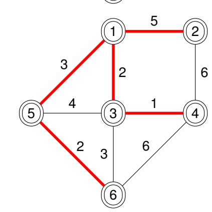
	dist	near
$\cancel{1}$	3	5
2	5	1
$\cancel{3}$	2	1
4	**1**	**3**
$\cancel{5}$	–	5
$\cancel{6}$	2	5

$\Rightarrow$



	dist	near
$\cancel{1}$	3	5
2	**5**	**1**
$\cancel{3}$	2	1
$\cancel{4}$	1	3
$\cancel{5}$	–	5
$\cancel{6}$	2	5

$\Rightarrow$



	dist	near
$\cancel{1}$	3	5
$\cancel{2}$	5	1
$\cancel{3}$	2	1
$\cancel{4}$	1	3
$\cancel{5}$	–	5
$\cancel{6}$	2	5

Prim's algorithm (cont'd)

```
algorithm Prim(G = (V, E, w), v_0 ∈ V):
    T ← ∅
    c ← 0
    for i = 0 to n do
        dist_i ← w(i, v_0)
        nearest_i ← v_0
    Heapify(dist)   (optional)
    for i = 1 to n − 1 do
        v ← DeQueue(dist)
        T ← T ∪ {(v, nearest_v)}
        c ← c + w((v, nearest_v))
        foreach neighbor x of v outside tree do
            if w(v, x) < dist_x then
                dist_x ← w(v, x)
                nearest_x ← v
                Update(dist_x)   (optional)
```

- Can organize *dist* as:
  - Heap: $O(n)$ to heapify and $O(\log n)$ to update but $O(1)$ to get the minimum
  - Plain array: no need to heapify or update, but $O(n)$ to get the minimum
- Running time ($|V| = n$, $|E| = m$):
  - The foreach loop runs $O(m)$ times overall (amortized)
  - Heap: $T(n) = n + n + n \log n + m \log n = O(m \log n)$
  - Array: $T(n) = n + n \times n + m = O(n^2)$

Kruskal and Prim (cont'd)

- Correctness of Prim:
  - Loop invariant: The partial tree is a minimum-cost spanning tree for the vertices it contains
- Comparison between Prim and Kruskal:

		Running time	Sparse graphs ($m = o(n^2/\log n)$)	Dense graphs ($m = O(n^2)$)
Kruskal		$O(m \log n)$	$O(n \log n)$	$O(n^2 \log n)$
Prim	Array	$O(n^2)$	$O(n^2)$	$O(n^2)$
	Heap	$O(m \log n)$	$O(n \log n)$	$O(n^2 \log n)$

  - No difference between Kruskal and Prim using a heap on sparse graphs
  - Notable advantage for Prim using an array on dense graphs

There is only one minimum spanning tree!

Lemma

If all the edge weights in a connected graph G are distinct then G has a unique minimum-cost spanning tree

- Proof by contrapositive:
  - Let T and T' be two minimum-cost spanning trees of G
  - Let e and e' be the minimum weight edge in $T \setminus T'$ and $T' \setminus T$ respectively, $w(e) \leq w(e')$
  - $T' \cup \{e\}$ must contain cycle C that goes through e, let $e'' \in C \setminus T$
  - It must be that $w(e'') \geq w(e') \geq w(e)$ (since $e'' \in T' \setminus T$)
  - Let $T'' = T' \cup \{e\} \setminus \{e''\}$ (greedy replace)
    - T'' is a spanning tree (we replaced one edge in a cycle with another in the same cycle)
    - $w(T'') = w(T') + w(e) - w(e'')$ so $w(T'') \leq w(T')$ (since $w(e) \leq w(e'')$)
    - But T' is a minimum-cost spanning tree, so it must be that $w(T'') = w(T')$ and so $w(e) = w(e'')$
- This kind of reasoning also works for not necessarily distinct edge weights as long as we use a consistent way of breaking ties

There is only one algorithm!

- Edge classification:
  - Useless: $(u, v) \notin F$ with u and v in the same connected component of F
  - Safe: minimum-weigth (u, v) with only u or v in a connected component of F
- Generic strategy for the minimum-cost spanning tree: Maintain an acyclic subgraph F of G such that F is a subgraph of the minimum-cost spanning tree of G by always choosing safe edges (and never useless edges)

Lemma

The minimum-cost spanning tree of G contains every safe edge

- Greedy-replace proof technique:
  - Show that the minimum-cost spanning tree of any $S \subseteq G$ contains the safe edge e for S
  - Let T be a minimum-cost spanning tree of G not containing e
  - It must have an edge e', $w(e') > w(e)$ that connects S with the rest of G
  - Then $T' = T \setminus \{e'\} \cup \{e\}$ is a spanning tree with $w(T') \leq w(T)$, a contradiction

Lemma

The minimum-cost spanning tree contains no useless edge

Single-source shortest path

- We are given a directed, weighted graph $G = (V, E, w)$
  - Notation: A path $p = \langle v_0, v_1, \ldots, v_k \rangle$ connects v_0 and v_1 and we write $v_0 \overset{p}{\leadsto} v_k$
  - The shortest-path weight from some vertex u to some vertex v is:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there exists a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

  - A shortest path from u to v is a path p such that $u \overset{p}{\leadsto} v$ and $w(p) = \delta(u, v)$
- When we are interested in finding shortest paths in a graph we solve a shortest-path problem
  - Single source, single destination (e.g., finding the shortest way to travel from point A to point B)
  - Single source, all destinations (e.g., broadcasting a message from one node in a network to all the other nodes)
  - All pairs shortest path (e.g., finding the fastest way to send information from any node in a network to any other node)

The single-source shortest-path problem

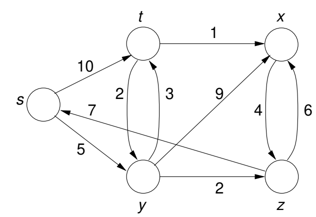
Lemma

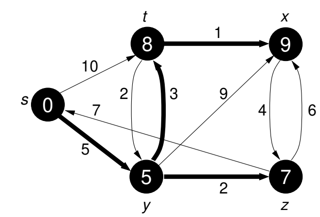
One shortest path contains other shortest paths within it. Formally, if $p = \langle v_0, v_1, \ldots, v_i, \ldots, v_j, \ldots, v_k \rangle$ is a shortest part from v_0 to v_k then the sub-path $\langle v_i, \ldots, v_j \rangle$ of p is a shortest path between v_i and v_j

- The lemma implies that the single source, single destination variant does not make sense since solving it effectively solves the single source, all destinations variant:

Input: a weighted graph G and a source node s:

Output: the shortest paths between s and any other vertex in G:





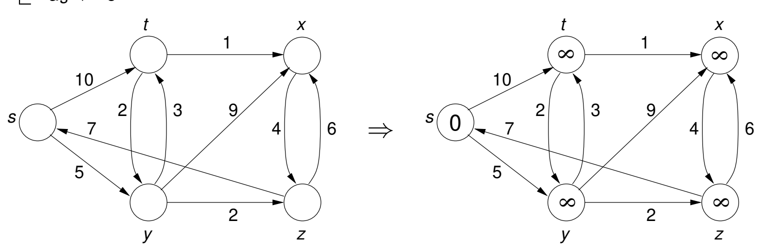
- The lemma also ensures that a greedy approach will work

Initialization

For each vertex v in the input graph, we keep two values:

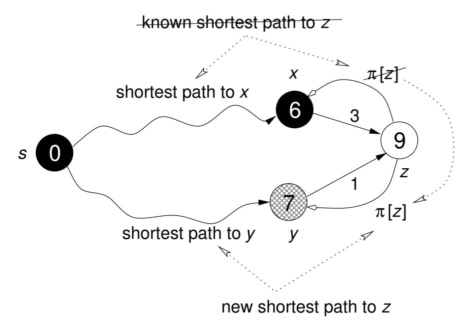
- d_v is a shortest-path estimate, initially ∞ for all the vertices but s
- π_v is the predecessor of v in the shortest path, initially NIL
  - our shortest path algorithm will set π_v for all the vertices in the graph
  - then, the predecessor link from some vertex v to s runs backwards along a shortest path from s to v

```
algorithm InitializeSingleSource(G = (V, E, w), s ∈ V; d, π):
    foreach v ∈ V do
        d_v ← ∞
        π_v ← Nil
    d_s ← 0
```



Relax!

- All algorithms that solve the shortest-path problem are built around the relaxation technique
- Simple idea: if we find something better, we go for it




```
algorithm Relax(y, z, w ∈ V; d, π):
    if d_z > d_y + w(y, z) then
        d_z ← d_y + w(y, z)
        DecreaseKey(Q, z, d_z)
        π_z ← y
```

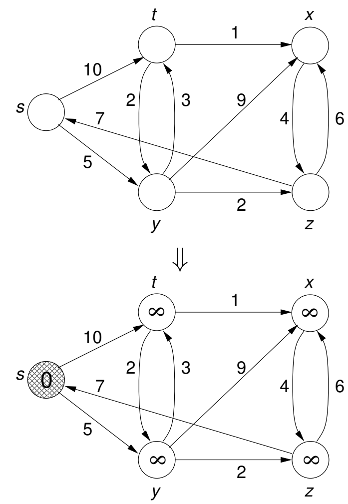
DIJKSTRA'S ALGORITHM

- Dijkstra's algorithm solves the single-source shortest-path problem on a weighted, directed graph $G = (V, E, w)$ with positive edge weights

algorithm DIJKSTRA($G = (V, E, w)$, $s \in V$; π):
  INITIALIZESINGLESOURCE(G, s)
  $S \leftarrow \emptyset$
  $Q \leftarrow$ MAKEQUEUE(V, d)
  while $\neg$ISEMPTY(Q) **do**
    $u \leftarrow$ DEQUEUE(Q)
    $S \leftarrow S \cup \{u\}$
    foreach v adjacent to u, $v \notin S$ **do**
      RELAX(u, v, w)

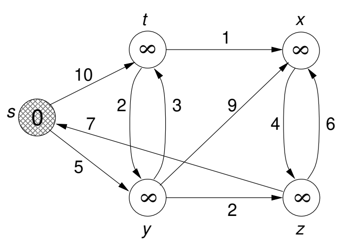
  - The algorithm maintains a set S of vertices whose final shortest path from the source s has been already determined
  - The algorithm (greedily) keeps selecting the most promising edge $u \in V \setminus S$, adds it to S, and relaxes all the edges leaving u
    - The "most promising" edge is the one with minimum d_u
    - Priority queue Q for quick access to this most promising edge

DIJKSTRA'S ALGORITHM (CONT'D)



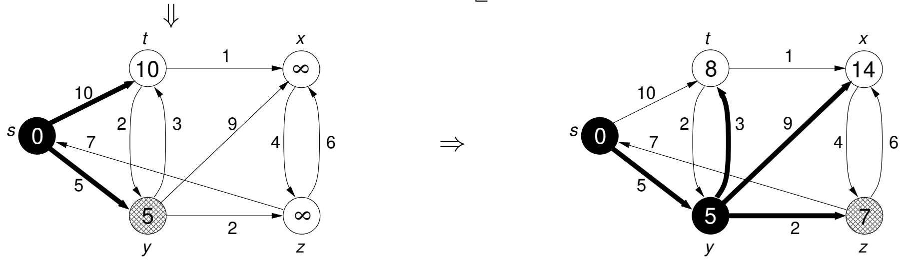
algorithm DIJKSTRA($G = (V, E, w)$, $s \in V$; π):
  INITIALIZESINGLESOURCE(G, s)
  $S \leftarrow \emptyset$
  $Q \leftarrow$ MAKEQUEUE(V, d)
  while $\neg$ISEMPTY(Q) **do**
    $u \leftarrow$ DEQUEUE(Q)
    $S \leftarrow S \cup \{u\}$
    foreach v adjacent to u, $v \notin S$ **do**
      RELAX(u, v, w)

DIJKSTRA'S ALGORITHM (CONT'D)

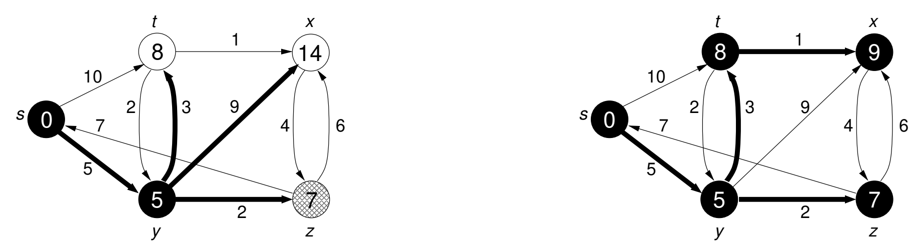


algorithm DIJKSTRA($G = (V, E, w)$, $s \in V$; π):
  INITIALIZESINGLESOURCE(G, s)
  $S \leftarrow \emptyset$
  $Q \leftarrow$ MAKEQUEUE(V, d)
  while $\neg$ISEMPTY(Q) **do**
    $u \leftarrow$ DEQUEUE(Q)
    $S \leftarrow S \cup \{u\}$
    foreach v adjacent to u, $v \notin S$ **do**
      RELAX(u, v, w)

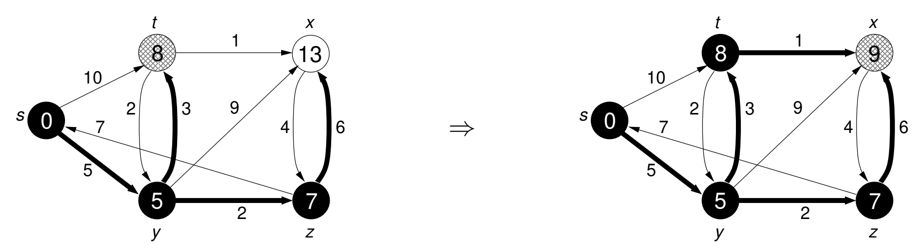
⇓



DIJKSTRA'S ALGORITHM (CONT'D)



⇓ **while** $\neg$ISEMPTY(Q) **do**. . . ⇑



Dijkstra's algorithm analysis

- Dijkstra's algorithm relies heavily of operations on the queue Q, namely EnQueue, DeQueue, and DecreaseKey, of running time, say, $t_+(n)$, $t_-(n)$, $t_x(n)$, respectively (with $n = |V|$, $m = |E|$)

algorithm Dijkstra($G = (V, E, w)$, $s \in V$; π):
  InitializeSingleSource(G, s)
  $S \leftarrow \emptyset$
  $Q \leftarrow$ MakeQueue(V, d)
  while $\neg$IsEmpty(Q) **do**
    $u \leftarrow$ DeQueue(Q)
    $S \leftarrow S \cup \{u\}$
    foreach v adjacent to u, $v \notin S$ **do**
      Relax(u, v, w)

- Total running time: $O(n \times t_+(n) + n \times t_-(n) + m \times t_x(n))$
- Correctness, or we always pick the right vertex: Let u_i and u_{i+1} be the vertices returned by two successive calls to DeQueue; then $d_{u_i} \leq d_{u_{i+1}}$ just after the extraction
  - Either $(u_i, u_{i+1}) \in E$ and u_{i+1} is relaxed, so $d_{u_{i+1}} = d_{u_i} + w((u_i, u_{i+1})) \geq d_{u_i}$
  - Or u_{i+1} is not relaxed so it is already in the queue so $d_{u_{i+1}} \geq d_{u_i}$
  - Trivial generalization for u_i and u_{i+k}
  - No vertex is dequeued more than once
  - Proof only works for positive edge weights

Analysis (cont'd)

- The performance of Dijkstra's algorithm depends heavily of how the priority queue is implemented (again!)

	$t_+(n)$	$t_-(n)$	$t_x(n)$
Array queue	$O(1)$	$O(n)$	$O(1)$
Heap queue	$O(\log n)$	$O(\log n)$	$O(\log n)$

	Running time	Sparse graphs ($m = o(n^2/\log n)$)	Dense graphs ($m = O(n^2)$)
Array Q	$O(n^2 + m)$	$O(n^2)$	$O(n^2)$
Heap Q	$O((n+m)\log n)$	$O(m \log n)$	$O(n^2 \log n)$

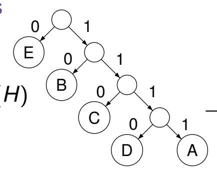
Data compression

- Represent data using the minimum amount of bits
- Lossy
  - Compressed data cannot be restored in its original form
  - Significant compression ratio
  - Mostly used for multimedia encoding
  - Examples: JPEG (Joint Photographic Experts Group) and MPEG (Moving Picture Experts Group)
- Lossless
  - Compressed data can be perfectly reconstructed
  - Lower compression ratio
  - Examples: Zip, Gif, Huffman encoding
- The Huffman code is an optimal variable-length prefix code
  - Minimizes the average number of bits/character based on the character frequencies of occurrence
  - Code system with the prefix property (prefix code): no code is a prefix of any other code
    - Necessary for decoding variable-length codes
    - Example: A, B, C, D can be encoded respectively as 0, 10, 110, 111, but not as 1, 10, 110, 111 (since the code for A would be a prefix for B, C and D)
    - Note in passing that fixed length codes (e.g. 00, 01, 10, 11) are all prefix codes

The Huffman code

- Example: Five characters with their frequency: A (5%), B (25%), C (20%), D (15%), E (35%)
  - Traditional (fixed-length encoding): A=000, B=001, C=010, D=011, E=100 (3 bits/character)
- Prefix code tree:
  - Choose and remove the letter with highest frequency, assign as left child
  - Repeat for the right child
  - Label left branches with 0 and right branches with 1
  - Code for a character is the path from root to letter

algorithm HuffmanLite(C):
  // C = set of n characters
  $H \leftarrow$ MakeQueue(C)
  $T \leftarrow$ new node
  for $i = 1$ **to** $n - 1$ **do**
    $T.left \leftarrow$ DeQueue(H)
    $T.right \leftarrow$ new node
    $T \leftarrow T.right$
  $T.right \leftarrow$ DeQueue(H)
  // Set codes in a BFS traversal



Letter	Freq	Code	Weighted # bits
A	0.05	1111	$4 \times 0.05 = 0.2$
B	0.25	10	$2 \times 0.25 = 0.5$
C	0.20	110	$3 \times 0.20 = 0.6$
D	0.15	1110	$4 \times 0.15 = 0.6$
E	0.35	0	$1 \times 0.35 = 0.35$

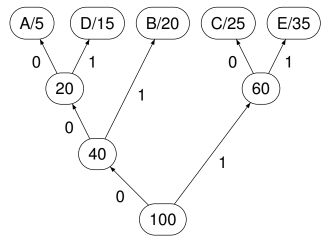
- Average bits per letter: 0.2+0.5+0.6+0.6+0.35=2.25
- Improvement of 25%

- Correctness: letters at different depths = different all-1 prefixes before 0
- Running time: $\Theta(n \log n)$ (both array and heap)

The Huffman code (cont'd)

- We can do better by assigning frequencies to internal nodes and choosing the best two frequencies to be the children of a new node:

algorithm HUFFMAN(C):
 // C = set of n characters
 $H \leftarrow$ MAKEQUEUE(C)
 for $i = 1$ **to** $n - 1$ **do**
  $T \leftarrow$ new node
  $T.left \leftarrow$ DEQUEUE(H)
  $T.right \leftarrow$ DEQUEUE(H)
  $T.freq \leftarrow T.left.freq + T.right.freq$
  INSERT($T freq$)
 // Set codes in a BFS traversal



- Running time: $\Theta(n^2)$ (sorted list) or $\Theta(n \log n)$ (heap)

Letter	Freq	Code	Weighted # bits
A	0.05	000	$3 \times 0.05 = 0.15$
B	0.25	10	$2 \times 0.25 = 0.5$
C	0.20	01	$2 \times 0.20 = 0.4$
D	0.15	001	$3 \times 0.15 = 0.45$
E	0.35	11	$2 \times 0.35 = 0.7$

- Average bits per letter: 2.2, 27% improvement
- Correctness: different paths ensure at least one different bit

Optimal text compression

- Huffman's algorithm produces an optimal tree
  - Show that the two least frequent characters have to be siblings in an optimal tree using a greedy-replace technique
  - Proceed upward by induction
  - See textbook
- Text compression algorithm:
  - Calculate the frequency of all letters in the text
  - Construct the Huffman tree
  - Encode all the text using the codes obtained from the Huffman tree
- Text recovery algorithm:
  - Traverse the Huffman tree from root to a leaf according to the input bits
  - Output the leaf label
  - Repeat traversal for as long as there are bits in the input
  - Note: this is why we need a code system with the prefix property!

The knapsack problem

- Given $w = \langle w_1, w_2, \ldots, w_n \rangle$ and $p = \langle p_1, p_2, \ldots, p_n \rangle$, find $x = \langle x_1, x_2, \ldots, x_n \rangle$ such that $\sum_{i=1}^{n} x_i p_i$ is maximized subject to $\sum_{i=1}^{n} x_i w_i \leq C$
  - Given n objects, each with a corresponding weight w_i and profit p_i and a knapsack of specific capacity C, choose the objects (or fractions) that you can fit in the knapsack so that the total profit is maximized
- Two versions:
  - Fractional knapsack: $0 \leq x_i \leq 1$
  - 0/1 knapsack: $x_i \in \{0, 1\}$

Fractional knapsack

- Greedy strategies:
  1. Take objects one at a time in increasing order of their weights, until the knapsack is full (a fraction may need to be taken for the last object)
  2. Take the objects in decreasing order of their profits
  3. Take the objects in decreasing order of their profits per unit weight ratio
- Example:

$$\begin{array}{rcllll} w & = & \langle & 5, & 10, & 20 & \rangle & C = 30 \\ p & = & \langle & 50, & 60, & 140 & \rangle \\ p/w & = & \langle & 10, & 6, & 7 & \rangle \end{array}$$

  1. $x = \langle 1, 1, 15/20 \rangle$, $P = 50 + 60 + 140 \times 15/20 = 215$
  2. $x = \langle 0, 1, 1 \rangle$, $P = 60 + 140 = 200$
  3. $x = \langle 1, 5/10, 1 \rangle$, $P = 50 + 60 \times 5/10 + 140 = 220$
- In fact it can be shown that the third strategy will always guarantee an optimal solution
  - Suppose that we have an optimal solution that uses some amount of the lower value density object
  - Then we substitute that with the same weight of the higher value density object and we obtain a better solution, a contradiction

0/1 knapsack

$$\begin{array}{rcl} w & = & \langle\ 5,\ 10,\ 20\ \rangle \\ p & = & \langle\ 50,\ 60,\ 140\ \rangle \\ p/w & = & \langle\ 10,\ 6,\ 7\ \rangle \\ C & = & 30 \end{array}$$

- By w: $x = \langle 1, 1, 0 \rangle$, $P = 110$
- By p: $x = \langle 0, 1, 1 \rangle$, $P = 200$
- By p/w: $x = \langle 1, 0, 1 \rangle$, $P = 190$

$$\begin{array}{rcl} w & = & \langle\ 18,\ 15,\ 10\ \rangle \\ p & = & \langle\ 25,\ 24,\ 15\ \rangle \\ p/w & = & \langle\ 1.38,\ 1.6,\ 1.5\ \rangle \\ C & = & 20 \end{array}$$

- By w: $x = \langle 0, 0, 1 \rangle$, $P = 15$
- By p: $x = \langle 1, 0, 0 \rangle$, $P = 25$
- By p/w: $x = \langle 0, 1, 0 \rangle$, $P = 24$

$$\begin{array}{rcl} w & = & \langle\ 5,\ 10,\ 20\ \rangle \\ p & = & \langle\ 80,\ 50,\ 120\ \rangle \\ p/w & = & \langle\ 16,\ 5,\ 6\ \rangle \\ C & = & 20 \end{array}$$

- By w: $x = \langle 1, 1, 0 \rangle$, $P = 130$
- By p: $x = \langle 0, 0, 1 \rangle$, $P = 120$
- By p/w: $x = \langle 1, 0, 0 \rangle$, $P = 50$

- No greey strategy guarantees an optimal solution for the 0/1 knapsack problem

The greedy-choice property

The greedy technique works only for those problems that have the greedy-choice property: We can assemble a globally optimal solution by making locally optimal (greedy) choices

- Goes hand in hand with the greedy-replace proof technique
- Many problems have the greedy-choice property, many more do not (such as the 0/1 knapsack)
- For some problems without the greedy-choice property may obtain a “good enough” solution for some reasonable definition of “good enough”
  - Good example: 0/1 knapsack
  - To be continued