Lecture 13, Tuesday 5/22/01

Greedy Algorithms

- Problem: set of n activities
  s_i, f_i start and end of activity i.
- i compatible with j if intervals do not intersect.
- Goal: find max # of compatible activities.

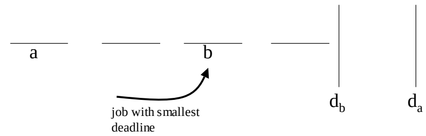
- Let k have smallest f_k and let A be OPT solution.
  Case 1: k in OPT. Claim: A-k is OPT for $S' = \{i \mid s_i > f_k\}$
  Assume not. Let B be OPT for S', |B|>|A|-1
  But then add k to B and we get better than A !
  Case 2: k not in A, finish time of 1st job in A is AFTER f_k
  ⟶ replace it with k !

  Thus we compute k, commit to it, compute S', and repeat !

109

Another greedy algorithm

- Task defined by (duration, deadline), eg. HW.
  Goal: find a schedule if one exists.
- Assume that there exists a schedule
  Claim: then there exists a schedule with
  first job = job with smallest deadline.



We can exchange b and a !!

110

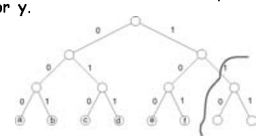
Summary

- Take locally best choice and commit to it.
- Main issue: proof that we can commit without loosing our chance
  to get an optimum solution.

111

Huffman encoding

- Idea: represent often encountered letters by shorter codes.
- Prefix code: a code for x is not a prefix for any code-word for y.

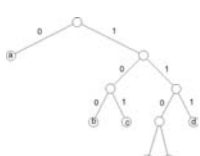


- In this example: c=010, e=100

112

Huffman encoding

- Assume that a is a very common symbol.



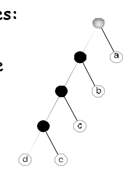
- Now: a = 0
  b = 100
  e = 1100

113

Huffman encoding

- Assume we know symbol frequencies:

50	40	5	3	2
a	b	c	d	e

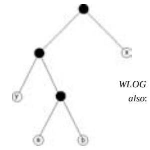


- 50*1+40*2+5*3+3*4+2*4 = 165, 1.65b/symbol
  instead of 3 !

114

Generating optimum encoding

- Claim: Let x&y be lowest freq. characters.
  Then there exists code where x&y *differ only in 1 bit.*



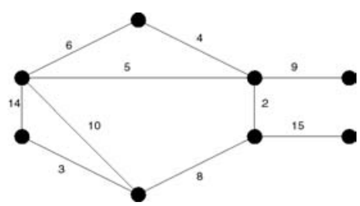
$$\left.\begin{array}{rl} WLOG & f(a)\le f(b), f(x)\le f(y) \\ also: & f(x)\le f(a), f(y)\le f(b) \end{array}\right\} \Rightarrow exchanging\ x \leftrightarrow a, y \leftrightarrow b \text{ can only help!}$$

- So does this mean that we do not need any other codes ?
  [Hint: consider a sequence 10101010101….]

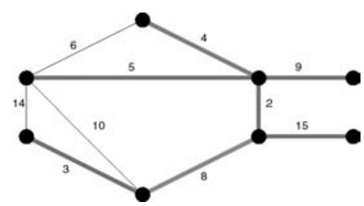
Min-Cost Spanning Tree

- Applications:
  - » Cable TV,
  - » VLSI,
  - » basic task for many optimization algs (eg. flow).
- Formally:
  - » Undirected graph *G=(V,E).*
  - » Weights $w: E \rightarrow R$
  - » *Goal*: find spanning tree of minimum weight.
    (spanning = connects all nodes in *G*)

Example

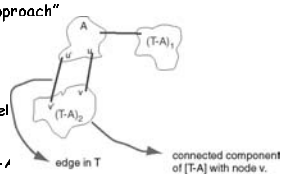


Example MST



Optimum Substructure

- Assume *T* is MST of *G*, $A \subseteq T$ subtree of T
  let uv be min-weight edge connecting A to (T-A)
  $\Rightarrow \exists$ MST T' s.t. $(A \cup uv) \subseteq T'$ (loose notation)
- Proof: "Cut-and-Paste" approach"
  - » Replace u'v' by uv !
- Questions:
  - » Why no more edges parallel to u'v' in T ?? - Cycles !
  - » Why u'v' exists at all ?? (walk in T until you hit [T-A



Prim's Algorithm

- Main idea:
  - » Pick a node v, set A={v}.
  - » Repeat:
    - find min-weight edge e, outgoing from A,
    - add e to A.
- Need support for finding an edge that is:
  - » outgoing,
  - » Min-weight among all outgoing.

Implementing Prim's Alg

- First try:
  - » Keep all edges (outgoing and internal) from *A* in a heap,
  - » new node: add all its edges to the heap.
  - » To get "next edge":
    - extract min-weight from heap
    - check if internal. (how ??)
    - if yes, discard and repeat.
- Time: *O(E)* insertions and *O(E)* deletions from heap:

  Total: *O(E log V)*

121

More about implementation

- Only *V-1* edges were used, the rest - wasted.
- Idea:
  - » keep nodes in the heap, instead of edges.
  - » Key: distance of node from *A* over a single edge.
  - » Initially: key(v) = infinity, for all v.
    key(root) = 0.

$x = root$
Repeat:
  $\forall v: vx \in E$ do:
    $key(v) = \min(key(v), w(vx))$
» Pick smallest-key x, add x to A.

- So why does this work ???

122

Alternative Implementations

- Total: O(E) decrease-key, O(V) extract-min.

	extract−min	decrease−key	Total
array	$O(V)$	$O(1)$	$O(V^2)$
heap	$O(\log V)$	$O(\log V)$	$O(E \log V)$
Fib. heap	$O(\log V)$	$O(1)$	$O(V \log V + E)$

123

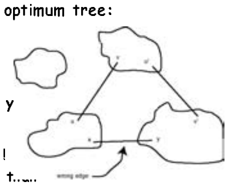
Kruskal's Algorithm

- Main loop:
  - » scan edges in increasing order of weight
  - » put edge in if no loop created.
- Why does this result in MST ??
  - » Observation: min-weight edge is always in MST.
    Proof: Assume there exists a tree without this edge.
    Add this edge to the tree - this creates a cycle.
    Delete max-weight edge on this cycle, we get a lighter tree !

124

Proof of Kruskal's algorithm

- Consider the instant when we are adding the first wrong edge,
  i.e. edge *xy* that is not in any optimum tree:



  - » blobs are current connected components.
  - » There exists a path from x to y in the optimum tree.
  - » uv and u'v' are not in our tree, thus they are heavier than xy !
  - » cut-and-paste to get a better t..... opt. tree: contradiction.

125

Implementation

- Given two nodes u and v, need to know if they are in the same connected component, i.e. in the same set.
  Find_Set(v)
- After adding edge uv, need to merge the set that includes u with the set that includes v.
  Union(Find_Set(u), Find_Set(v))
- Total: O(V) Make_Set
  O(E) Find_Set
  O(V) Union
- Section 22.4 explains how to achieve these ops in α(E,V) time, where α is inverse Ackerman function.
  (Union-Find data structure)
- $\alpha(m,n)<5$ for $m,n < 10^8$!!!

126

Simple Union-Find Implementation

- Main idea:
  - » Maintain every set as a linked list, every element points to head of the list.
  - » Merge smaller lists into larger ones.
- Work:
  - » Find-Set takes O(1).
  - » Union: O(1) per element of the smaller list.
  - » Each time an element is charged during union, his set at least doubles !! O(log V) charges per element for all unions. →
  - » Total: O(V log V) work for all Unions.
- Total time: Sort + O(E+V log V)

127

Lecture 16, Tuesday 11/28/00

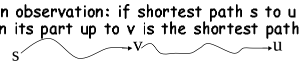
Dynamic Programming

- Main problem with greedy approaches: sometimes we can not commit up-front.
- Dynamic programming:
  - » Meta-technique, not a specific algorithm.
- Main idea:
  - » solve many small sub-problems,
  - » combine solution to several small subproblems to solve larger subproblems.
  - » continue combining until we solve the original problem.

128

Single-Source Shortest Paths

- Read Chapter 25.
- Problem:
  - » Directed graph $G=(V,E)$, n nodes, m edges.
  - » Edge uv has (real) weight w(uv).
  - » Distinguished node *s*, the "source".
  - » Need to find shortest path from s to all nodes reachable from *s*.
- Main observation: if shortest path s to u goes through v, then its part up to v is the shortest path from s to v.



129

Bellman-Ford

- Dynamic programming:
  - » Subproblem: $d^k(v)$ = distance from s to v in up to *k* "hops".
  - » To reach v in at most k+1 hops:
    - – reach neighbor of v in at most k hops,
    - – hop to v
    - – alternatively, reach v in at most k-1 hops
    - – phase k computes $d^k(v)$ for all v.

$$d^{k+1}(v) = \min\left\{d^k(v),\ \min\left\{d^k(u)+w(uv) \mid uv \in E\right\}\right\}$$

  - » Terminates in n-1 phases if no negative cycles
    Proof in the book.
    (Main idea: if more than n-1 hops, the path is not simple.)

Total time = O(nm)

130

Bellman-Ford

- Early termination:
  We can terminate at phase k if, for all v, $d^k(v) = d^{k-1}(v)$,
  since no more changes will happen in $d^k(v)$ for larger values of k.
  (might terminate earlier than after n-1 phases)
- If negative cycle exists then no termination, even in n-1 phases:
  Proof:
  - – Consider edge v_iv_{i+1} along the cycle at termination.
  - – If terminated, then for all edges on the cycle: $d(v_i) \le d(v_{i+1}) + w(v_iv_{i+1})$
  - – Sum up: $\sum_i d(v_i) \le \sum_i d(v_i) +$ (weight of the cycle) $\Rightarrow$ weight of the cycle ≥ 0

131

Another example: Matrix chain multiplication

- Consider the following chain: $A_1 \times A_2 \times \cdots \times A_n$, A_1 is $[p_o \times p_1]$, A_2 is $[p_1 \times p_2]$, etc.

$$\left[A_1 \times A_2\right]_{i,j} = \sum_{k=1}^{p_1} A_1[i,k]A_2[k,j], \quad \text{time} \approx p_0 p_1 p_2$$

- Example: [5x100] [100x2] [2x50]
  - » Multiplying last two and then by the first one:
    100x2x50 + 5x100x50 = 35,000 multiplications.
  - » Multiplying first two and then by the last one:
    5x100x2 + 5x2x50 = 1500
- Order of multiplication affects the amount of work !

132

Solving matrix chain multiplication

- Observation:
  - » Consider last optimum multiplication: $(A_1 \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_n)$
  - » Then both $(A_1 \times \cdots \times A_k)$ and $(A_{k+1} \times \cdots \times A_n)$ were computed optimally !! (Why ??)
- Subproblems: $m(i,j)$ is best "time" to multiply $(A_i \times \cdots \times A_j)$

$$m(i,j) = \begin{cases} 0 & if\ i = j \\ \min_{i \le k < j} \{ m(i,k) + m(k+1,j) + p_{i-1} p_k p_j \} & if\ i < j \end{cases}$$

- Answer is *m(1,n)*
- Why can't we just use as subproblems the time to multiply matrices 1 to i ??

133

Lecture 17, Thursday 11/30/00

Matrix chain continued

- Lets try to analyze using recurrence relation:

$$T(1) \ge 1$$
$$T(n) \ge 1 + \sum_{k=1}^{n-1} [T(k) + T(n-k) + 1] \ge 2 \sum_{k=1}^{n-1} T(k) + n$$

by substitution, easy to see that $T(n) \ge 2^{n-1}$

- Wrong approach ! There are only $O(n^2)$ different subproblems !
- Build the table bottom up, for increasing *(j-I)*.
- $O(n)$ per each $m(i,j)$, total $O(n^3)$.

134

Summary - Dynamic Programming

- Find optimum substructure
- Define subproblems (not too many of them !)
- Organize subproblems into a table.
- Make sure there is a way to fill the table.

135

Longest common-subsequence

- Consider two sequences:

```
x = A  B  C  B  D  A  B        |x| = m
     /  |  \  |
y = B  D  C  A  B   A          |y| = n
```

- Greedy: does not work ! (Why ??)
- Brute force: take any substring of x, check against y. Total: $O(2^m n)$, too slow !

136

Optimum Substructure

- Define subproblems: $C(i,j) = LCS(x_1, x_2, \ldots, x_i, y_1, y_2, \ldots, y_j)$
- Observe that C(m,n) is the answer that we seek.
- Theorem:

$$C(i,j) = \begin{cases} C(i-1, j-1) + 1 & \text{if } x_i = y_i \\ \max\{C(i, j-1), C(i-1, j)\} & \text{otherwise} \end{cases}$$

Proof: Case 1, $x_i = y_j$.
Consider $z_1, \ldots z_k$ LCS of $(x_1 \ldots x_i)$, $(y_1 \ldots y_j)$
if $z_k \neq x_i$, then z is not LCS !!! (Why ??)

Now we claim that $z_1 \ldots z_{k-1}$ is LCS of $(x_1 \ldots x_{i-1})$, $(y_1 \ldots y_{j-1})$

Proof: if there is a longer than Z sequence, just extend it !

137

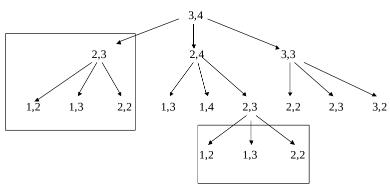
Proof: continued

- Case 2: $x_i \neq y_i$
  either: $z_k = x_i$ (2a)
  or $z_k = y_j$ (2b)
  or not equal to either of them. (2c)
  - » Case 2a: $z_k = x_i \Rightarrow z_k \neq y_j \Rightarrow z_1, \ldots z_k$ is a LCS of $(x_1 \ldots x_i)$, $(y_1 \ldots y_{j-1})$ (Why ??)
  - » Case 2b is symmetric.
  - » Do Case 2c at home.

138

Recursive algorithm

- We can use the theorem to construct a recursive algorithm.
  Consider its tree:



Analysis

- Depth of the tree is O(m+n), leads to $O(3^{m+n})$ bound, too large !
- Main idea: we see repeating sub-question,
  only O(mn) different ones !
- memoization: after computing sub-problem answer, remember it.
  dynamic programming: compute the table bottom-up.

Computing the table

- Fill the table starting from top-left corner, and going row-by-row:

	y_i	B	D	C	A	B	A
x_i							
		0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

- Each element depends on the one above, one left, and if $x_i = y_i$,
  then it is one more then the diagonal up-left element.

Knapsack Problem

- Problem statement:
  - » We have n items, *I*-th item costs *v(I)* and weights *w(I)*.
  - » We have a knapsack that can hold total *W* weight.
  - » Goal: maximize total value of items that we choose to put into the knapsack, without exceeding total allowed weight W.
- Abstraction of many real problems:
  from investing to telephone call routing.
- Fractional (allowed to take part of an item)- easy !
  do greedy, choose best value-per-weight element.

Fractional vs. Integer Knapsack

- Consider the following example:

w	\$	\$/W
20	30	1.5
50	60	1.2
50	50	1

  - » **Greedy: #1+#2 gives $90**
  - » **Optimum: #2+#3, gives $110**
  - » **Fractional: #1 +#2 + 3/5 of #3, gives $120.**
- Optimum substructure:

  Consider optimum solution: $x_1, x_2, \ldots, x_k$,
  where $x_i = 0$ means we do not take the item, and $x_i = 1$ means we take it.

  Claim: $x_1, x_2, \ldots, x_{k-1}$ is optimum for $\underbrace{S - x_k}, \underbrace{W - w(x_k)}$.

  set of items
  knapsack size (max weight)

Solving Knapsack

- Subproblems:

  $C(i,w)$ - OPT solution using items 1 to i, knapsack w.

$$C(i,w) = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max\{\underbrace{v_i + C(i-1, w-w_i)}_{\text{take i-th}}, \underbrace{C(i-1,w)}_{\text{do not take i-th}}\} & \text{otherwise} \end{cases}$$

- Table size in nW, O(1) per element, TOTAL = O(nW)
- But knapsack is NP-Hard !
  Do we indeed have a contradiction here ??

  No contradiction since <u>W is not polynomial</u> in the size of the input...

Lecture 18, Tuesday 12/5/00

Graph Algorithms

- Examples of graph problems:
  - » Direct applications:
    - – City streets map: reachability, shortest path, congestion management
    - – Communication networks: planning, fault tolerance/reliability, topology augmentation
  - » Indirect applications:
    - – Assigning MDs to hospitals
    - – Scheduling jobs on a multiprocessor
    - – Searching solution spaces
- Restate as a graph problem → solve → map back

Depth First Search

- Visit(u)

```
Visit(u)
      color(u) = gray;   d(u) = time;  time++;
      for each neighbor  w of v:
               if w is white then Visit(w)
       color(u) = black;  f(u) = time;  time++;
```

- Initially, set all nodes white, examine nodes one-by-one, call Visit if node is still white.
- Node visited once, edge touched twice: Running time O(n+m)
- At home: Read theorems 23.6 and 23.8 ! (we will only sketch the proofs)

Edge Classification

- Classification of uw according to (color of u) -> (color of w): (when the edge is considered)
  - » Tree edge: gray -> white
  - » Back edge: gray -> gray
  - » Forward: gray -> black, u ancestor of w.
  - » Cross: other gray -> black edges.
- How to distinguish forward and cross edges ??
  We can use d() time !

Parenthesis Theorem

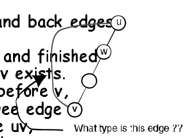
- Theorem:
  For any two nodes u and v,
  the two intervals [d(u),f(u)] and [d(v),f(v)] either:
  - » Do not intersect, or
  - » [d(u),f(u)] includes [d(v),f(v)] , v descendant of u, or
  - » [d(v),f(v)] includes [d(u),f(u)] , u descendant of v.
- Proof:
  - » Assume (wlog) d(u)< d(v).
  - » If v was not discovered before finishing u, then we have case 1 above.
  - » If v was discovered, then we have to finish it before returning and finishing u, leading to case 2.
  - » Case 3 is symmetric.

White-Path Lemma

- In (directed or undirected) graph G, node v is descendant of u iff at d(u) (time when u was discovered) there is a path from v to u using only currently white nodes.
- Proof:
  - » Assume v is descendant of u.
    Let ww' be edge on the u->v path in the tree.
    If w' was not white at d(u), then ww' will not be tree edge.
    Thus, all nodes on the u->v path are white when u is discovered.
  - » Assume that at d(u) there is a white path from u to v.
    Let ww' be the first edge on this path, with w' closest to u
    so that w is descendant of u but w' is not.
    - – We have f(u)> f(w) > d(w) > d(u).
    - – But we have to discover w' after starting u and before finishing w:
      d(u) < d(w') < f(w) < f(u)
    - – By parenthesis theorem, w' is also a descendant of u, contradiction.

Simple Lemma

- Lemma: if G <u>undirected</u>, then only tree and back edges
  Proof: wlog, d(u)<d(v).
  Thus v must be discovered and finished before finishing u, since uv exists.
  If uv discovered from u, before v, it is tree edge
  if v was discovered before uv, uv becomes a back edge.



- Why does the proof break down in the <u>directed</u> case ?

Discovering Cycles

- Claim: G acyclic iff DFS yields no back edges.
- Proof:
  - » Trivial to observe that back edge implies a cycle.
  - » Assume there exists a cycle:
    - – Let v be the node with smallest d on the cycle and let uv be edge of the cycle.
    - – At d(v) all nodes on the cycle, including u, are white.
    - – All these nodes, including u, become descendants of v.
    - – Thus, when u is scanned, we will discover uv edge and mark it as "back edge".

151

Topological Sort

- Directed acyclic graph G.
- Algorithm:
  - » Call DFS to compute finishing times f[v] for each vertex v.
  - » As each v is finished, insert it onto the front of linked list
  - » Return the linked list.
- Claim: the output list is a legal topological sort.
  - » Sufficient to prove that, for every u and v s.t. (uv) is an edge, we have f[v] < f[u]. (Why ??)
  - » Consider edge (uv) explored by DFS.
    Observe that when (uv) is explored, v can not be gray !
    (back edge implies cycle)
  - » If v white, it becomes descendant of u, and thus f[v] < f[u].
  - » If v black, it finished before u started, so again f[v] < f[u].

152

Back to shortest paths: Dijkstra's Algorithm

- We can do better than Bellman-Ford if no negative-weight edges !
- Algorithm:

$d(s) = 0;\quad \forall v \neq s: d(v) = \infty;$
Construct heap, key(v) = $d(v)$;
While heap not empty:
  $u = \text{extract\_min(heap)}$;
  for each v s.t. $uv \in E$:
   if $d(v) > d(u) + w(uv)$
   then $d(v) = d(u) + w(uv)$

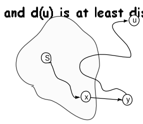
- Main idea: add node with shortest perceived distance.
- Time: n extract_min, m decrease_key
  binary heap: O(m log n)
  Fib. Heap: O(m+n log n)

153

Correctness of Dijkstra's algorithm

- Correctness Proof:
  - » Let u be the first extracted node with d(v) not equal to distance. (note that once v is extracted, its d(v) is not adjusted)
  - » Consider shortest path s to u, focus on edge (xy) where x was extracted already (its d(x) is correct distance) and y was not yet extracted. (Why does such edge exist ?)
  - » Observe that d(y) is at most d(x)+w(xy), since x was already processed.
  - » All distances are non-negative and d(u) is at least dist(s,u):

$$\begin{aligned} d(u) &\geq dist(s,u) \\ &= dist(s,x) + w(xy) + dist(y,u) \\ &= d(x) + w(xy) + dist(y,u) \\ &\geq d(y) + dist(y,u) \\ &\geq d(y) \end{aligned}$$



  - » Thus d(u) is currently not minimum and u will not be extracted !

154

END

155