CSC 373: Algorithm Design and Analysis
Lecture 4

Allan Borodin

January 14, 2013

Lecture 4: Outline (for this lecture and next lecture)

- Some concluding comments on optimality of EST Greedy Interval Colouring Algorithm
- Interval Graphs
- Graph MIS and graph colouring
- Kruskal's MST
- Huffman coding
- Greedy algorithms for makespan problem

Comments on the proof technique in graph colouring

- The proof technique used in proving the optimality of the EST greedy algorithm for interval colouring is also often used for proving approximations.
- The idea is to find some bound (or bounds) that any solution must satisfy and then relate that to the algorithm's solution.
- In this case, consider the maximum number of intervals in the input set that intersect at any given point.

Observation

The number of colours must be at least this large.

- For the interval colouring proof, it then just remained to show that the greedy algorithm will never use more than this number of colours.

Why doesn't the Greedy Colouring Algorithm exceed this intrinsic bound?

- Recall that we have sorted the intervals by nondecreasing starting time (i.e. earliest start time first).
- Let k = maximum number of intervals in the input set that intersect at any given point.
- Suppose for a contradiction that

the algorithm used more than k colours.

- Consider the first time (say on some interval ℓ) that the greedy algorithm would have used $k+1$ of colours.
  - Then it must be that there are k intervals intersecting ℓ.
  - Let s_ℓ be the starting time of interval I_ℓ.
  - These intersecting intervals must all include s_ℓ. Why?
  - Hence, there are $k+1$ intervals intersecting at s_ℓ!

Interval graphs

- As we remarked last class, there is a natural way to view the interval scheduling and colouring problems as **graph problems**.
- Let $\mathcal{I}$ be a set of intervals. We can construct the intersection graph $G(\mathcal{I}) = (V, E)$ where
  - $V = \mathcal{I}$
  - (u, v) is an edge in E iff the intervals corresponding to u and v intersect.
- Any graph that is the intersection graph of a set of intervals is called an interval graph.

Graph MIS and Colouring

- Let $G = (V, E)$ be a graph.
- The following two problems are known to be "NP hard to approximate" (to within a factor of $n^{1-\epsilon}$ for any $\epsilon > 0$) for arbitrary graphs:

Graph MIS

- A subset U of V is an independent set (aka stable set) in G if for all $u, v \in U$, (u, v) is not an edge in E.
- The maximum independent set (MIS) problem is to find a maximum size independent set U.

Graph colouring

- A function c mapping vertices to $\{1, 2, \ldots, k\}$ is a valid colouring of G if $c(u)$ is not equal to $c(v)$ for all $(u, v) \in E$.
- The graph colouring problem is to find a valid colouring so as to minimize the number of colours k.

Efficient algorithms for interval graphs

- Given a set $\mathcal{I}$ of intervals, it is easy to construct its intersection graph $G(\mathcal{I})$.

Note

Given any graph G, there is a linear-time algorithm to decide if G is an interval graph and if so to construct an interval representation.

- The interval scheduling (resp. interval colouring) problem becomes the graph MIS (resp. colouring) problem for the intersection graph and hence these problems are efficiently solved for interval graphs.
  - Question: Is there a graph theoretic explanation?
  - YES: interval graphs are chordal graphs.
- The minimum colouring number (chromatic number) of a graph is always at least the size of a maximum clique (clique number).
- The greedy interval colouring proof shows that for interval graphs (and chordal graphs) the chromatic number = clique number.; i.e. perfect graphs. However, Mycielskis Theorem shows that there exist triangle-free graphs with arbitrarily high chromatic number.

Greedy algorithms for the MST problem

- We will start with Kruskal's algorithm. The presentation in DPV also considers appropriate data structures for implementing the algorithm.
- In terms of the basic structure of the algorithm it is very similar to the EFT algorithm for interval scheduling.

Kruskal's algorithm

Order edges so that $w_{e_1} \leq w_{e_2} \leq \ldots \leq w_{e_m}$.
Let $T := \emptyset$ % T is the current forest to be extended to an MST
For $i : 1, \ldots, m$
  If e_i connects two components of T:
    $T := T \cup \{e_i\}$
End For

Claim

Same style inductive proof (using cut property to show T_i is promising) could be used to show Kruskal'a algorithm is optimal.

Code for Kruskal's algorithm as in DPV text

DPV Figure 5.4: Kruskals minimum spanning tree algorithm

Procedure *Kruskal*(G, w)
Input: A connected undirected graph $G = (V, E)$ with edge weights w_e
Output: A minimum spanning tree defined by the edges X
For all $u \in V$:
    makeset(u)
$X = \emptyset$
Sort the edges E by (non-decreasing) weight
For all edges $\{u, v\} \in E$, in increasing order of weight
    If *find*$(u) \neq$ *find*(v):
        add edge $\{u, v\}$ to X
    union(u, v)

Comment

The inductive proof for optimality of Kruskal or Prim's MST algorithm shows that when all edges are distinct, the MST is unique.

Prim's MST and Dikstra's Least Cost Paths

- Prim's algorithm (and the proof of its optimality) for the MST problem is very similar but now the next edge is adaptively chosen to be the smallest edge leaving the current component.

- The style of Prim's MST algorithm is very similar to Dikstra's algorithm for computing least cost paths from a single source node s to all the other nodes in a directed graph with non-negative edge costs.

- We can view the single source least cost problem as computing a least cost tree with root s.
  - At each iteration i, having computed the tree for nodes in some set S_i, the next edge (or node) to be chosen is the one that minimizes the cost to a node not in S_i by adding an edge leaving S_i.
  - It can be shown (in some precise model for greedy algorithms) that an adaptive order for choosing edges is necessary; that is, a fixed order will not work.

Huffman (prefix-free) binary encoding

- Consider a set of symbols $\Gamma = \{s_1, s_2, \ldots, s_n\}$.
- These symbols appear in some context (e.g. words in a document, discrete samples from a signal, etc.).
- We want to encode each symbol s_i as a binary string, call it σ_i.
- We assume that these symbols occur with different frequencies with symbol s_i having frequency f_i.
- Clearly if a symbol say s occurs very often (resp. infrequently), we want to use a relatively short (resp. long) string to represent it.
- In order to simplify decoding, a nice property is that the encodings $\{\sigma_i\}$ satisfy the prefix-free property that no codeword σ_i is the prefix of another code word σ_j.

Prefix binary codes as binary trees

- Such an encoding is equivalent to a full ordered binary tree T; that is, a rooted binary tree where
  - Every non leaf has exactly two children
  - With the left edge say labeled 0 and the right edge labeled 1
  - With every leaf labeled by a symbol in Γ
- Then the labels along the path to a leaf define the string encoding the symbol at that leaf. The goal is to create such a tree T so as to minimize

$$cost(T) = \sum_i f_i \cdot (\text{depth of } s_i \text{ in } T)$$

- Equivalently we are minimizing the expected symbol length, namely

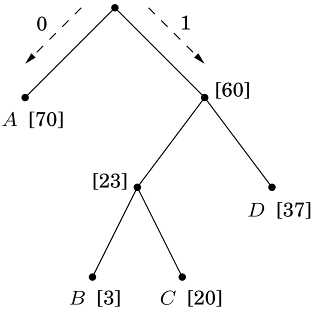
$$\mathbf{E}_{s \in \Gamma}[|\sigma(s)|] = \sum_i p_i \cdot (\text{depth of } s_i \text{ in } T)$$

  where $p_i = \frac{f_i}{\sum_i f_i}$ is the probability of s_i.
- The intuitive idea is to greedily combine the two lowest frequency symbols s_1 and s_2 to create a new symbol with frequency $f_1 + f_2$.

An example of Huffman coding in DPV

Figure 5.10 A prefix-free encoding. Frequencies are shown in square brackets.

Symbol	Codeword
A	0
B	100
C	101
D	11



The Huffman algorithm as in DPV text

Code for Huffman coding

Procedure *Huffman*($\mathbf{f}$)
Input: An array $\mathbf{f}[1 \ldots n]$ of frequencies with $f_1 \leq f_2 \ldots \leq f_n$
Output: An encoding tree with n leaves
Let H be a priority queue of integers, ordered by $\mathbf{f}$
For $i : 1, \ldots, n$
  $insert(H, i)$
For $k : n+1, \ldots, 2n-1$
  $i = deletemin(H)$, $j = deletemin(H)$
  create a node numbered k with children i, j
  $f[k] = f[i] + f[j]$
  $insert(H, k)$

What are chordal graphs?

The following two slides were not discussed in class but I leave them here for anyone who is interested.

- There are many equivalent ways to define chordal graphs.
- For our purposes, let's define chordal graphs $G = (V, E)$ as those having a perfect elimination ordering (PEO) of the vertices.

PEO

An ordering $v(1), v(2), \ldots, v(n)$ such that for all i,

Neighbourhood($v(i)$) intersect $\{v(i+1), \ldots, v(n)\}$ is a clique (i.e. the MIS of the induced graph of the inductive neighbourhood is 1).

Note

- Ordering intervals by earliest finishing times will provide a PEO for the intersection graph of intervals
- Hence interval graphs are chordal.

More on chordal graphs

- We can abstract the arguments used for **interval selection** to show the optimality of greedy algorithms for any chordal graph using a **PEO ordering**.
- We can abstract the arguments used for **interval colouring** to show the optimality of greedy algorithms for any chordal graph using a **reverse PEO ordering**.
- An equivalent (and initial) definition of chordal graphs are

graphs which do not have any k-cycles (for $k > 3$) as induced subgraphs

- What are and are not chordal graphs? For example a 4-cycle cannot be an interval graph. Trees are chordal graphs.
- Can generalize chordal graphs by generalizing PEO orderings so that the induced neighbourhoods have MIS $= k$ for some small k.