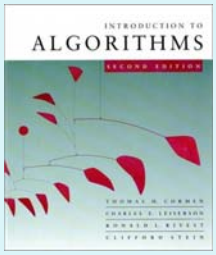
Algorithms

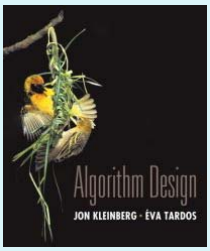
Richard Anderson
University of Washington

Today's topics

- Teaching Algorithms
- Active Learning in Algorithms
- Big Ideas: Solving Problems in Practice
- Mysore / Theory Discussion

Text books





University of Washington Course

CSE 421 Introduction to Algorithms (3)
Techniques for design of efficient algorithms. Methods for showing lower bounds on computational complexity. Particular algorithms for sorting, searching, set manipulation, arithmetic, graph problems, pattern matching. Prerequisite: CSE 322; CSE 326.

- Algorithm Design, by Jon Kleinberg and Eva Tardos, 2005.
- Ten week term
  - 3 lectures per week (50 minutes)
  - Midterm, Final

Course overview

- Stable Marriage (2)
- Basic Graph Algorithms (3)
- Greedy Algorithms (2)
- Graph Algorithms (4)
- Divide and Conquer and Recurrences (5)
- Dynamic Programming (5)
- Network Flow and Applications (5)
- NP Completeness (3)

Analyzing the course and content

- What is the purpose of each unit?
  - Long term impact on students
- What are the learning goals of each unit?
  - How are they evaluated
- What strategies can be used to make material relevant and interesting?
- How does the context impact the content

Overall course context

- Senior level elective
  - Students are not required to take this class
  - Approximately half the students take this course
  - Theory course: no expectation of programming
- Data structures is a pre-requisite
- Little coordination with data structures course
  - Some overlap in material
  - Generally different instructors
- Text book highly regarded by faculty
- Course is “algorithms by techniques”

Stable Marriage

- Very interesting choice for start of the course
- Stable Marriage is a non-standard topic for the class
- Advanced algorithm to start the class with new ideas
- Show a series of different algorithmic techniques

All of Computer Science is the Study of Algorithms

How to study algorithms

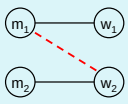
- Zoology
- Mine is faster than yours is
- Algorithmic ideas
  - Where algorithms apply
  - What makes an algorithm work
  - Algorithmic thinking

Introductory Problem: Stable Matching

- Setting:
  - Assign TAs to Instructors
  - Avoid having TAs and Instructors wanting changes
    - E.g., Prof A. would rather have student X than her current TA, and student X would rather work for Prof A. than his current instructor.

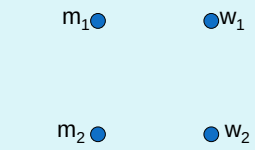
Formal notions

- Perfect matching
- Ranked preference lists
- Stability



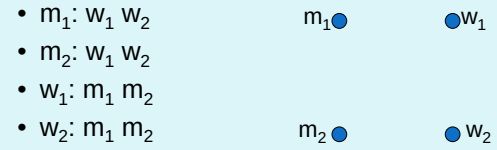
Example (1 of 3)

- m_1: w_1 w_2
- m_2: w_2 w_1
- w_1: m_1 m_2
- w_2: m_2 m_1



Example (2 of 3)

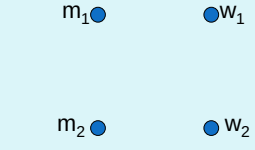
- m_1: w_1 w_2
- m_2: w_1 w_2
- w_1: m_1 m_2
- w_2: m_1 m_2



Find a stable matching

Example (3 of 3)

- m_1: w_1 w_2
- m_2: w_2 w_1
- w_1: m_2 m_1
- w_2: m_1 m_2

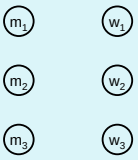


A closer look

- Stable matchings are not necessarily fair

m_1: w_1 w_2 w_3
m_2: w_2 w_3 w_1
m_3: w_3 w_1 w_2

w_1: m_2 m_3 m_1
w_2: m_3 m_1 m_2
w_3: m_1 m_2 m_3

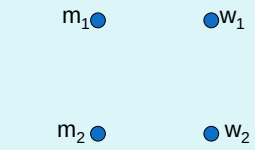


How many stable matchings can you find?

Example

m_1: w_1 w_2 w_3
m_2: w_1 w_3 w_2
m_3: w_1 w_2 w_3

w_1: m_2 m_3 m_1
w_2: m_3 m_1 m_2
w_3: m_3 m_1 m_2



Intuitive Idea for an Algorithm

- m proposes to w
  - If w is unmatched, w accepts
  - If w is matched to m_2
    - If w prefers m to m_2, w accepts
    - If w prefers m_2 to m, w rejects
- Unmatched m proposes to highest w on its preference list that m has not already proposed to

Algorithm

Initially all m in M and w in W are free
While there is a free m
    w highest on m's list that m has not proposed to
    if w is free, then match (m, w)
    else
        suppose (m_2, w) is matched
        if w prefers m to m_2
            unmatch (m_2, w)
            match (m, w)

Does this work?

- Does it terminate?
- Is the result a stable matching?
- Begin by identifying invariants and measures of progress
  - m's proposals get worse
  - Once w is matched, w stays matched
  - w's partners get better (have lower w-rank)

Claim: The algorithm stops in at most n^2 steps

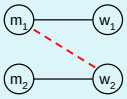
- Why?

When the algorithms halts, every w is matched

- Why?
- Hence, the algorithm finds a perfect matching

The resulting matching is stable

- Suppose
  - m_1 prefers w_2 to w_1
- How could this happen?



Result

- Simple, $O(n^2)$ algorithm to compute a stable matching
- Corollary
  - A stable matching always exists

Basic Graph Algorithms

- This material is necessary review
- Terminology varies so cover it again
- Formal setting for the course revisited
  - Big Oh notation again
- Debatable on how much depth to go into formal proofs on simple algorithms

Polynomial time efficiency

- An algorithm is efficient if it has a polynomial run time
- Run time as a function of problem size
  - Run time: count number of instructions executed on an underlying model of computation
  - T(n): maximum run time for all problems of size at most n
- Why Polynomial Time?
  - Generally, polynomial time seems to capture the algorithms which are efficient in practice
  - The class of polynomial time algorithms has many good, mathematical properties

Ignoring constant factors

- Express run time as O(f(n))
- Emphasize algorithms with slower growth rates
- Fundamental idea in the study of algorithms
- Basis of Tarjan/Hopcroft Turing Award

Formalizing growth rates

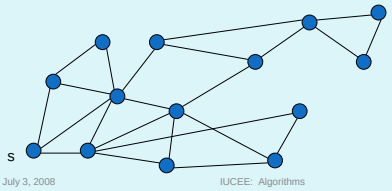
- T(n) is O(f(n)) $[T : Z^+ \rightarrow R^+]$
  - If n is sufficiently large, T(n) is bounded by a constant multiple of f(n)
  - Exist c, n_0, such that for $n > n_0$, $T(n) < c\ f(n)$
- T(n) is O(f(n)) will be written as: T(n) = O(f(n))
  - Be careful with this notation

Graph Theory

- G = (V, E)
  - V – vertices
  - E – edges
- Undirected graphs
  - Edges sets of two vertices {u, v}
- Directed graphs
  - Edges ordered pairs (u, v)
- Many other flavors
  - Edge / vertices weights
  - Parallel edges
  - Self loops

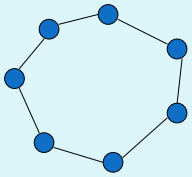
Breadth first search

- Explore vertices in layers
  - s in layer 1
  - Neighbors of s in layer 2
  - Neighbors of layer 2 in layer 3 . . .



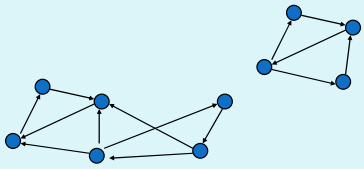
Testing Bipartiteness

- If a graph contains an odd cycle, it is not bipartite



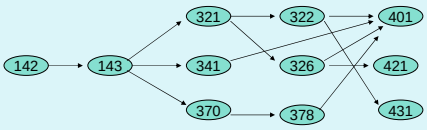
Directed Graphs

- A Strongly Connected Component is a subset of the vertices with paths between every pair of vertices.



Topological Sort

- Given a set of tasks with precedence constraints, find a linear order of the tasks



Greedy Algorithms

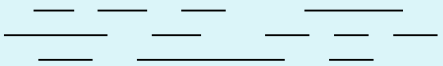
- Introduce an algorithmic paradigm
- Its hard to give a formal definition of greedy algorithms
- Proof techniques are important
  - Need to formally prove that these things work
- New material to students

Greedy Algorithms

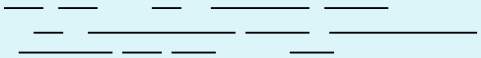
- Solve problems with the simplest possible algorithm
- The hard part: showing that something simple actually works
- Pseudo-definition
  - An algorithm is Greedy if it builds its solution by adding elements one at a time using a simple rule

Greedy solution based on earliest finishing time

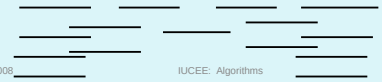
Example 1



Example 2

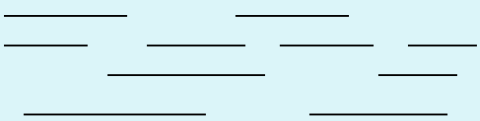


Example 3



Scheduling all intervals

- Minimize number of processors to schedule all intervals



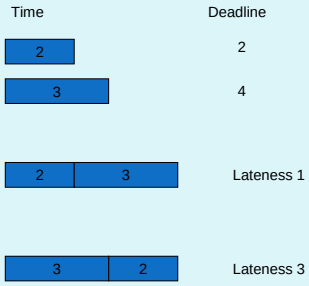
Algorithm

- Sort by start times
- Suppose maximum depth is d, create d slots
- Schedule items in increasing order, assign each item to an open slot
- Correctness proof: When we reach an item, we always have an open slot

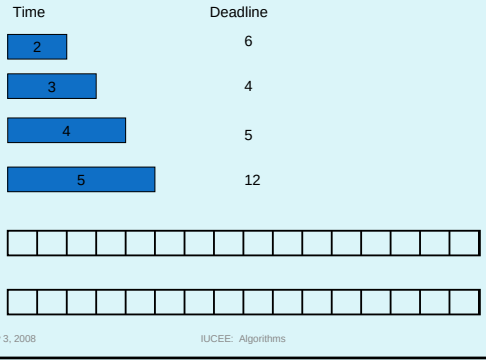
Scheduling tasks

- Each task has a length t_i and a deadline d_i
- All tasks are available at the start
- One task may be worked on at a time
- All tasks must be completed
- Goal minimize maximum lateness
  - Lateness = $f_i - d_i$ if $f_i >= d_i$

Example



Determine the minimum lateness



Homework Scheduling

- Tasks to perform
- Deadlines on the tasks
- Freedom to schedule tasks in any order

Greedy Algorithm

- Earliest deadline first
- Order jobs by deadline

- This algorithm is optimal

Analysis

- Suppose the jobs are ordered by deadlines, $d_1 <= d_2 <= \ldots <= d_n$
- A schedule has an *inversion* if job j is scheduled before i where j > i

- The schedule A computed by the greedy algorithm has no inversions.
- Let O be the optimal schedule, we want to show that A has the same maximum lateness as O

Shortest Paths and MST

- These graph algorithms are presented in the framework of greedy algorithms
- Students will have seen the algorithms previously
- Attempt is made to have students really understand the proofs
  - Classical results

Assume all edges have non-negative cost

Dijkstra's Algorithm

S = {}; d[s] = 0; d[v] = infinity for v != s

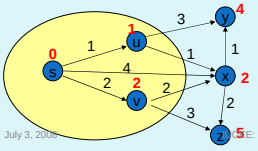
While S != V

Choose v in V-S with minimum d[v]

Add v to S

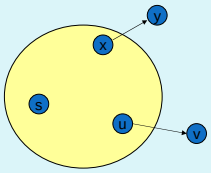
For each w in the neighborhood of v

d[w] = min(d[w], d[v] + c(v, w))



Proof

- Let v be a vertex in V-S with minimum d[v]
- Let P_v be a path of length d[v], with an edge (u,v)
- Let P be some other path to v. Suppose P first leaves S on the edge (x, y)
  - $P = P_{sx} + c(x,y) + P_{yv}$
  - $Len(P_{sx}) + c(x,y) >= d[y]$
  - $Len(P_{yv}) >= 0$
  - $Len(P) >= d[y] + 0 >= d[v]$

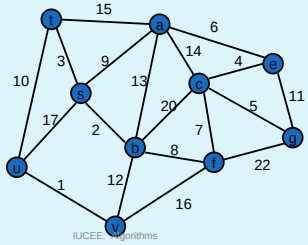


http://www.cs.utexas.edu/users/EWD/

- Edsger Wybe Dijkstra was one of the most influential members of computing science's founding generation. Among the domains in which his scientific contributions are fundamental are
  - algorithm design
  - programming languages
  - program design
  - operating systems
  - distributed processing
  - formal specification and verification
  - design of mathematical arguments

Greedy Algorithm 1
Prim's Algorithm

- Extend a tree by including the cheapest out going edge

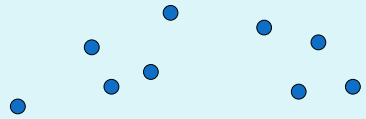


Construct the MST with Prim's algorithm starting from vertex a

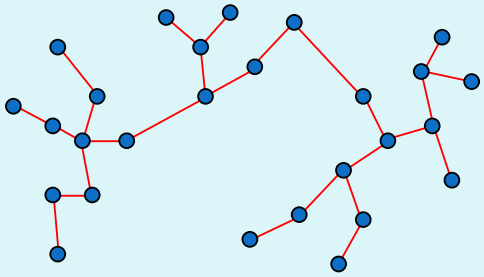
Label the edges in order of insertion

Application: Clustering

- Given a collection of points in an r-dimensional space, and an integer K, divide the points into K sets that are closest together



K-clustering



Recurrences

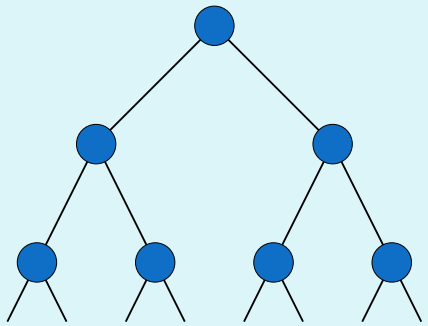
- Big question on how much depth to cover recurrences
  - Full mathematical coverage
  - Intuition
- Students have little background on recurrences coming in
  - Generally not covered in earlier courses
- My emphasis is in conveying the intuition
  - Students can look up the formulas when they need them

T(n) <= 2T(n/2) + cn; T(2) <= c;

Recurrence Analysis

- Solution methods
  - Unrolling recurrence
  - Guess and verify
  - Plugging in to a "Master Theorem"

Unrolling the recurrence



Recurrences

- Three basic behaviors
  - Dominated by initial case
  - Dominated by base case
  - All cases equal – we care about the depth

Recurrence Examples

- T(n) = 2 T(n/2) + cn
  - O(n log n)
- T(n) = T(n/2) + cn
  - O(n)

- More useful facts:
  - $\log_k n = \log_2 n / \log_2 k$
  - $k^{\log n} = n^{\log k}$

T(n) = aT(n/b) + f(n)

What you really need to know about recurrences

- Work per level changes geometrically with the level
- Geometrically increasing (x > 1)
  - The bottom level wins
- Geometrically decreasing (x < 1)
  - The top level wins
- Balanced (x = 1)
  - Equal contribution

Strassen's Algorithm

Multiply 2 x 2 Matrices:

$$\begin{vmatrix} r & s \\ t & u \end{vmatrix} = \begin{vmatrix} a & b \\ c & d \end{vmatrix} \begin{vmatrix} e & g \\ f & h \end{vmatrix}$$

$r = p_1 + p_4 – p_5 + p_7$

$s = p_3 + p_5$

$t = p_2 + p_5$

$u = p_1 + p_3 – p_2 + p_7$

Where:

$p_1 = (b + d)(f + g)$

$p_2 = (c + d)e$

$p_3 = a(g – h)$

$p_4 = d(f – e)$

$p_5 = (a – b)h$

$p_6 = (c – d)(e + g)$

$p_7 = (b – d)(f + h)$

Divide and Conquer

- Classical algorithmic technique
- This is the texts weak point
- Students are probably already familiar with the sorting algorithms
- Lectures generally show off classical results
- FFT is a very hard result for the students
  - CSE students have little to tie it to

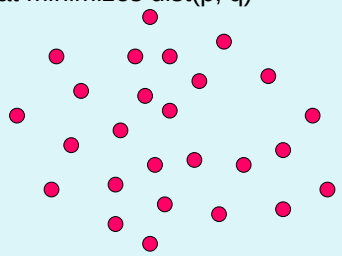
Divide and Conquer Algorithms

- Split into sub problems
- Recursively solve the problem
- Combine solutions

- Make progress in the split and combine stages
  - Quicksort – progress made at the split step
  - Mergesort – progress made at the combine step

Closest Pair Problem

- Given a set of points find the pair of points p, q that minimizes dist(p, q)



Karatsuba's Algorithm

Multiply n-digit integers x and y

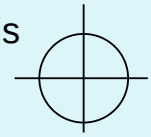
Let $x = x_1 2^{n/2} + x_0$ and $y = y_1 2^{n/2} + y_0$
Recursively compute
$a = x_1 y_1$
$b = x_0 y_0$
$p = (x_1 + x_0)(y_1 + y_0)$
Return $a2^n + (p - a - b)2^{n/2} + b$

Recurrence: $T(n) = 3T(n/2) + cn$

FFT, Convolution and Polynomial Multiplication

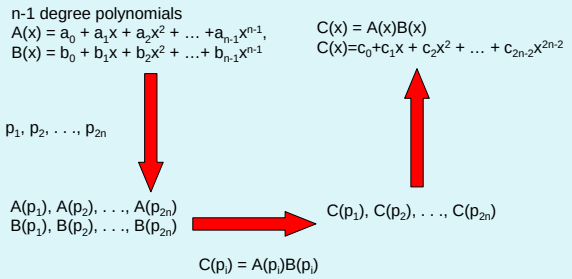
- Preview
  - FFT - O(n log n) algorithm
    - Evaluate a polynomial of degree n at n points in O(n log n) time
  - Computation of Convolution and Polynomial Multiplication (in O(n log n)) time

Complex Analysis



- Polar coordinates: $re^{\theta i}$
- $e^{\theta i} = \cos\theta + i \sin\theta$
- a is a n^{th} root of unity if $a^n = 1$
- Square roots of unity: +1, -1
- Fourth roots of unity: +1, -1, i, -i
- Eighth roots of unity: +1, -1, i, -i, $\beta + i\beta$, $\beta - i\beta$, $-\beta + i\beta$, $-\beta - i\beta$ where β = sqrt(2)

Polynomial Multiplication



FFT Algorithm

// Evaluate the 2n-1th degree polynomial A at
// $\omega_{0,2n}, \omega_{1,2n}, \omega_{2,2n}, \ldots, \omega_{2n-1,2n}$
FFT(A, 2n)
    Recursively compute FFT(A_{even}, n)
    Recursively compute FFT(A_{odd}, n)

    for j = 0 to 2n-1
        $A(\omega_{j,2n}) = A_{even}(\omega^2_{j,2n}) + \omega_{j,2n}A_{odd}(\omega^2_{j,2n})$

Dynamic Programming

- I consider this to be the most important part of the course
- Goal is for them to be able to apply this technique to new problems
- Key concepts need to be highlighted so students start to see the structure of dynamic programming solutions

Dynamic Programming

- The most important algorithmic technique covered in CSE 421
- Key ideas
  - Express solution in terms of a polynomial number of sub problems
  - Order sub problems to avoid recomputation

Subset Sum Problem

- Let $w_1,\ldots,w_n$ = {6, 8, 9, 11, 13, 16, 18, 24}
- Find a subset that has as large a sum as possible, without exceeding 50

Subset Sum Recurrence

- Opt[j, K] the largest subset of {w_1, …, w_j} that sums to at most K

Subset Sum Grid

Opt[j, K] = max(Opt[j – 1, K], Opt[j – 1, K – w_j] + w_j)

4																
3																
2																
1																
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

{2, 4, 7, 10}

Knapsack Problem

- Items have weights and values
- The problem is to maximize total value subject to a bound on weght
- Items {I_1, I_2, … I_n}
  - Weights {w_1, w_2, …,w_n}
  - Values {v_1, v_2, …, v_n}
  - Bound K
- Find set S of indices to:
  - Maximize $\sum_{i \in S} v_i$ such that $\sum_{i \in S} w_i$ <= K

Knapsack Recurrence

Subset Sum Recurrence:

Opt[j, K] = max(Opt[j – 1, K], Opt[j – 1, K – w_j] + w_j)

Knapsack Recurrence:

Knapsack Grid

Opt[j, K] = max(Opt[j – 1, K], Opt[j – 1, K – w_j] + v_j)

4																
3																
2																
1																
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Weights {2, 4, 7, 10} Values: {3, 5, 9, 16}

Optimal line breaking and hyphen-ation

- Problem: break lines and insert hyphens to make lines as balanced as possible
- Typographical considerations:
  - Avoid excessive white space
  - Limit number of hyphens
  - Avoid widows and orphans
  - Etc.

Longest Common Subsequence

- Application of dynamic programming
- LCS is one of the classic DP algorithms
- Space efficiency discussed
  - Space more expensive than time
  - If we just want the length of the string, O(n) space is easy
  - Very clever algorithm allows reconstruction of LCS in O(n) space as well
  - Included as an advanced topic

Longest Common Subsequence

- $C=c_1 \ldots c_g$ is a subsequence of $A=a_1 \ldots a_m$ if C can be obtained by removing elements from A (but retaining order)
- LCS(A, B): A maximum length sequence that is a subsequence of both A and B

ocurranec attacggct

occurrence tacgacca

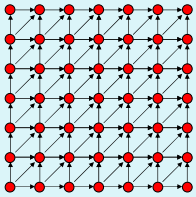
LCS Optimization

- $A = a_1 a_2 \ldots a_m$
- $B = b_1 b_2 \ldots b_n$
- Opt[j, k] is the length of $LCS(a_1 a_2 \ldots a_j, b_1 b_2 \ldots b_k)$

If $a_j = b_k$, Opt[j,k] = 1 + Opt[j-1, k-1]

If $a_j \mathrel{!}= b_k$, Opt[j,k] = max(Opt[j-1,k], Opt[j,k-1])

Dynamic Programming Computation



How good is this algorithm?

- Is it feasible to compute the LCS of two strings of length 100,000 on a standard desktop PC? Why or why not.

Algorithm Performance

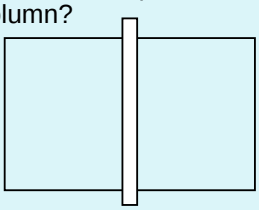
- O(nm) time and O(nm) space
- On current desktop machines
  - n, m < 10,000 is easy
  - n, m > 1,000,000 is prohibitive
- Space is more likely to be the bounding resource than time

Computing LCS in O(nm) time and O(n+m) space

- Divide and conquer algorithm
- Recomputing values used to save space

Divide and Conquer Algorithm

- Where does the best path cross the middle column?



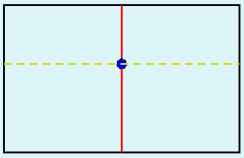
- For a fixed i, and for each j, compute the LCS that has a_i matched with b_j

Divide and Conquer

- $A = a_1,\ldots,a_m$ $B = b_1,\ldots,b_n$
- Find j such that
  - LCS($a_1 \ldots a_{m/2}$, $b_1 \ldots b_j$) and
  - LCS($a_{m/2+1} \ldots a_m$,$b_{j+1} \ldots b_n$) yield optimal solution
- Recurse

Algorithm Analysis

- $T(m,n) = T(m/2, j) + T(m/2, n-j) + cnm$



Shortest Paths

- Shortest paths revisited from the dynamic programming perspective
  - Dynamic programming needed if edges have negative cost

Shortest Path Problem

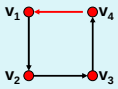
- Dijkstra's Single Source Shortest Paths Algorithm
  - O(mlog n) time, positive cost edges
- General case – handling negative edges
- If there exists a negative cost cycle, the shortest path is not defined
- Bellman-Ford Algorithm
  - O(mn) time for graphs with negative cost edges

Shortest paths with a fixed number of edges

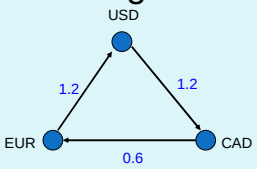
- Find the shortest path from v to w with exactly k edges
- Express as a recurrence
  - $Opt_k(w) = \min_x [Opt_{k-1}(x) + c_{xw}]$
  - $Opt_0(w) = 0$ if v=w and infinity otherwise

If the pointer graph has a cycle, then the graph has a negative cost cycle

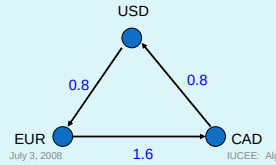
- If P[w] = x then M[w] >= M[x] + cost(x,w)
  - Equal when w is updated
  - M[x] could be reduced after update
- Let $v_1, v_2, \ldots v_k$ be a cycle in the pointer graph with (v_k, v_1) the last edge added
  - Just before the update
    - $M[v_j] >= M[v_{j+1}] + cost(v_{j+1}, v_j)$ for $j < k$
    - $M[v_k] > M[v_1] + cost(v_1, v_k)$
  - Adding everything up
    - $0 > cost(v_1,v_2) + cost(v_2,v_3) + \ldots + cost(v_k, v_1)$



Foreign Exchange Arbitrage



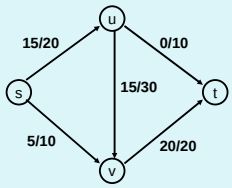
	USD	EUR	CAD
USD	------	0.8	1.2
EUR	1.2	------	1.6
CAD	0.8	0.6	-----

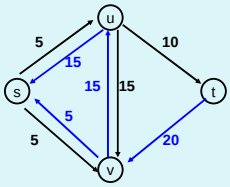


Network Flow

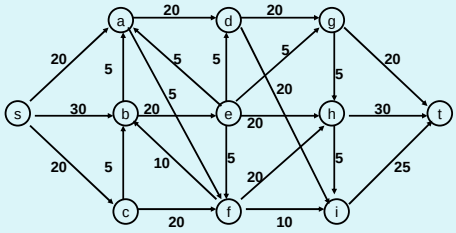
- This topic move the course into combinatorial optimization
- Key is to understand what the network flow problem is, and the basic combinatorial theory behind it
  - Many more sophisticated algorithms not covered

Flow assignment and the residual graph





Find a maximum flow



Ford-Fulkerson Algorithm (1956)

while not done

Construct residual graph G_R

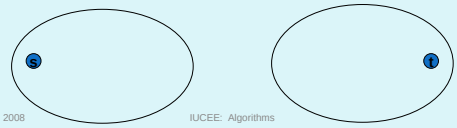
Find an s-t path P in G_R with capacity b > 0

Add b units along in G

If the sum of the capacities of edges leaving S is at most C, then the algorithm takes at most C iterations

MaxFlow – MinCut Theorem

- There exists a flow which has the same value of the minimum cut
- Proof: Consider a flow where the residual graph has no s-t path with positive capacity
- Let S be the set of vertices in G_R reachable from s with paths of positive capacity



Network Flow Applications

- Applications of network flow are very powerful
- Problems that look very unlike flow can be converted to network flow
- Brings up the theme of problem mapping

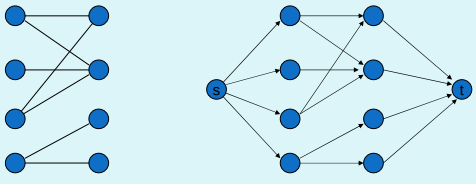
Problem Reduction

- Reduce Problem A to Problem B
  - Convert an instance of Problem A to an instance Problem B
  - Use a solution of Problem B to get a solution to Problem A
- Practical
  - Use a program for Problem B to solve Problem A
- Theoretical
  - Show that Problem B is at least as hard as Problem A

Bipartite Matching

- A graph G=(V,E) is bipartite if the vertices can be partitioned into disjoints sets X,Y
- A matching M is a subset of the edges that does not share any vertices
- Find a matching as large as possible

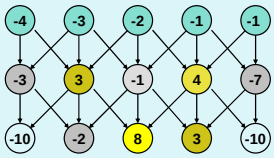
Converting Matching to Network Flow



Open Pit Mining

- Each unit of earth has a profit (possibly negative)
- Getting to the ore below the surface requires removing the dirt above
- Test drilling gives reasonable estimates of costs
- Plan an optimal mining operation

Mine Graph



Setting the costs

- If p(v) > 0,
  - cap(v,t) = p(v)
  - cap(s,v) = 0
- If p(v) < 0
  - cap(s,v) = -p(v)
  - cap(v,t) = 0
- If p(v) = 0
  - cap(s,v) = 0
  - cap(v,t) = 0

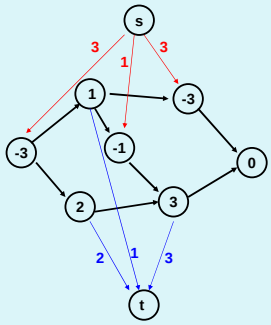


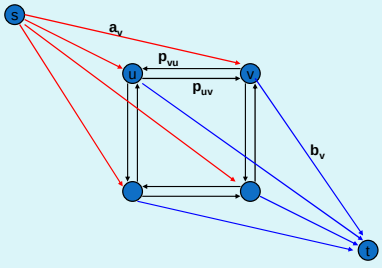
Image Segmentation

- Separate foreground from background

Image analysis

- a_i: value of assigning pixel i to the foreground
- b_i: value of assigning pixel i to the background
- p_{ij}: penalty for assigning i to the foreground, j to the background or vice versa
- A: foreground, B: background
- $Q(A,B) = \Sigma_{\{i \text{ in } A\}} a_i + \Sigma_{\{j \text{ in } B\}} b_j - \Sigma_{\{(i,j) \text{ in } E,\ i \text{ in } A,\ j \text{ in } B\}} p_{ij}$

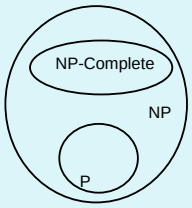
Mincut Construction



NP Completeness

- Theory topic from the algorithmic perspective
- Students will see different aspects of NP-Completeness in other courses
- Complexity theory course will prove Cook's theorem
- The basic goal is to remind students of specific NP complete problems
- Material is not covered in much depth because of the "last week of the term" problem

Theory of NP-Completeness The Universe



What is NP?

- Problems solvable in non-deterministic polynomial time . . .
- Problems where "yes" instances have polynomial time checkable certificates

NP-Completeness

- A problem X is NP-complete if
  - X is in NP
  - For every Y in NP, $Y <_P X$
- X is a "hardest" problem in NP
- If X is NP-Complete, Z is in NP and $X <_P Z$
  - Then Z is NP-Complete

History

- Jack Edmonds
  - Identified NP
- Steve Cook
  - Cook's Theorem – NP-Completeness
- Dick Karp
  - Identified "standard" collection of NP-Complete Problems
- Leonid Levin
  - Independent discovery of NP-Completeness in USSR

Populating the NP-Completeness Universe

- Circuit Sat $<_P$ 3-SAT
- 3-SAT $<_P$ Independent Set
- 3-SAT $<_P$ Vertex Cover
- Independent Set $<_P$ Clique
- 3-SAT $<_P$ Hamiltonian Circuit
- Hamiltonian Circuit $<_P$ Traveling Salesman
- 3-SAT $<_P$ Integer Linear Programming
- 3-SAT $<_P$ Graph Coloring
- 3-SAT $<_P$ Subset Sum
- Subset Sum $<_P$ Scheduling with Release times and deadlines

Find a satisfying truth assignment

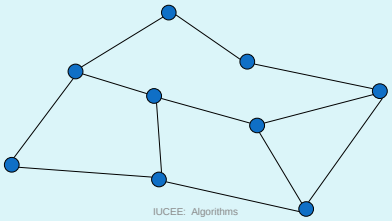
(x || y || z) && (!x || !y || !z) && (!x || y) && (x || !y) && (y || !z) && (!y || z)

IS $<_P$ VC

- Lemma: A set S is independent iff V-S is a vertex cover
- To reduce IS to VC, we show that we can determine if a graph has an independent set of size K by testing for a Vertex cover of size n - K

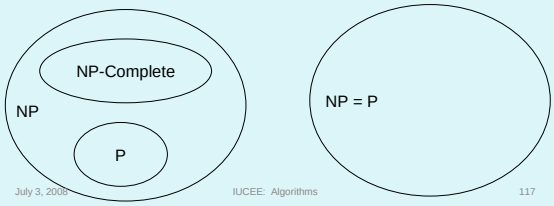
Graph Coloring

- NP-Complete
  - Graph K-coloring
  - Graph 3-coloring
- Polynomial
  - Graph 2-Coloring



What we don't know

- P vs. NP



What about 'negative instances'

- How do you show that a graph does not have a Hamiltonian Circuit
- How do you show that a formula is not satisfiable?

What about 'negative instances'

- How do you show that a graph does not have a Hamiltonian Circuit
- How do you show that a formula is not satisfiable?