CS 124 **Final Exam Review Notes** 5/6/2017

Contents

1 Notes about the Final

- Material covered in the first half of the class will be on the exam (e.g., dynamic programming), but the final will emphasize the second half of the course.
- These review notes are not comprehensive. Material that does not appear here is still fair game.
- Averages in past years have been around 50%. Don't be discouraged if you don't solve every problem!

2 Study Tips

In general, as you've seen on the midterm, you can probably break down the types of questions you'll be asked into two broad categories: those that test you on more on specific algorithms we've covered, and those that test you on your ability to apply more general algorithmic tools/paradigms. When studying specific algorithms:

- Understand the main idea behind the algorithm, the runtime, and the space complexity for any algorithm that we've covered.
  - Trying to boil down what the algorithm is doing to a 1-3 sentence summary can help to ensure you understand the key point of the algorithm.
  - For space and time complexity, you want to be able to identify what the most expensive operations are or what the recursion leading to the complexity is.
- Understand the constraints for what types of problems that algorithm solves.
  - In a similar vein, you want to practice applying algorithms to problems, and understanding what the key features are of a problem solved by an algorithm helps you figure out which algorithm to use.

    - For example, if we have an algorithm on DAGs, do you know why it wouldn't work on graphs with cycles? (Try thinking about what would happen to the algorithm if you invalidated one of the assumptions; what step(s) of the analysis would go wrong in this case? Could you adjust the algorithm to fix them?)

- Focus on the big ideas of the analysis of the algorithms above all the details.
    - A good example is Linear Programming; you'll probably get more utility from understanding what kind of problem it solves (maximizing/minimizing a linear equation based on linear constraints) and what kind of algorithm solves the problem (Simplex Method) than to memorize how to actually solve it.
- Try thinking about variations of the algorithms. For example, if you have a major step in the algorithm, try thinking about what would happen if tweaked that step. Alternatively, if the algorithm uses one data structure, what would happen if you replaced it with a different one?

When studying more general tools/paradigms:

- Practice applying them.
    - Apart from the problems we've released, you can find others in the textbook and online.
    - One thing that's helpful here is to also solve problems where you don't know what tool you're supposed to be using, so that you can practice choosing them.
- Try to break down the process into pieces (e.g. the three steps of dynamic programming, the steps for showing NP-completeness).
- Understand what the key characteristics are that make a tool work well.
    - For example, dynamic programming tends to work better when you can break your problem down into slightly smaller subproblems (say one size smaller), whereas divide and conquer works when you can break your problem down into significantly smaller problems (say half the size).
- When you're working on the problems, a lot of times the most difficult piece is not the details of analysis/proof, but the initial choices you make in set.
    - For dynamic programming and divide-and-conquer, the first step is to choose a subproblem. However, a lot of times the difficulty in getting to the answer is not in the analysis after this point, but in choosing the subproblem itself (so if you find yourself getting stuck, try using a different subproblem).
    - For dealing with P/NP, when doing reductions you generally want to choose a problem that's as close to the original as possible; choosing a subproblem that's further away will probably force you to do a lot of extra work when you're trying to prove the reduction.

3 Mathematics

3.1 Big-O and Master Theorem

Big-O notation is a way to describe the rate of growth of functions. In CS, we use it to describe properties of algorithms (number of steps to compute or amount of memory required) as the size of the inputs to the algorithm increase.

$f(n)$ is $O(g(n))$	if there exist c, N such that $f(n) \leq c \cdot g(n)$ for all $n \geq N$.	f "$\leq$" g
$f(n)$ is $o(g(n))$	if $\lim_{n\to\infty} f(n)/g(n) = 0$.	f "$<$" g
$f(n)$ is $\Theta(g(n))$	if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$.	f "$=$" g
$f(n)$ is $\Omega(g(n))$	if there exist c, N such that $f(n) \geq c \cdot g(n)$ for all $n \geq N$.	f "$\geq$" g
$f(n)$ is $\omega(g(n))$	if $\lim_{n\to\infty} g(n)/f(n) = 0$.	f "$>$" g

A very useful tool when trying to find the asymptotic rate of growth (Θ) of functions given by recurrences is the Master Theorem. The solution to the recurrence relation $T(n) = aT(n/b) + cn^k$, where $a \geq 1$, $b \geq 2$ are integers, and c and k are positive constants, satisfies

$$T(n) \text{ is } \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k \end{cases}$$

3.2 Probability

A discrete random variable X which could take the values in some set S can be described by the probabilities that it is equal to any particular $s \in S$ (which we write as $P(X = s)$). Its *expected value* $\mathbb{E}(X)$ is the "average" value it takes on, i.e. $\sum_{s\in S} s \cdot \mathbb{P}(X = s)$.

A few useful facts:

- If some event happens with probability p, the expected number of independent tries we need to make in order to get that event to occur is $1/p$.
- $\displaystyle\sum_{i=0}^{\infty} p^i = \frac{1}{1-p}$ (for $|p| < 1$) and $\displaystyle\sum_{i=0}^{n} p^i = \frac{1-p^{n+1}}{1-p}$ (for all p)
- For a random variable X taking on non-negative integer values, $\mathbb{E}(X) = \displaystyle\sum_{k=0}^{\infty} \mathbb{P}(X > k)$.

Two useful tools:

- *Linearity of expectation*: for any random variables X, Y, $\mathbb{E}(aX + bY) = a\mathbb{E}(X) + b\mathbb{E}(Y)$. In particular, this holds even if X and Y are not independent (for example, if $X = Y$).
- *Markov's inequality*: for any non-negative random variable X and $\lambda > 0$, $\mathbb{P}(X > \lambda \cdot \mathbb{E}(X)) < \frac{1}{\lambda}$. In other words, this indicates that the probability of X being significantly larger than its expectation is small.

4 Deterministic Data Structures

4.1 Heap

A *Heap* is a data structure to enable fast deletion of the maximal or minimal element in a dynamic list. Consequently, we often use them to implement priority queues. The following are operations and runtimes for a binary `Max-Heap`:

- `Build-Heap`: Turn an (unordered) list of elements into a heap in time $O(n)$
- `Max`: Identify the maximal element in time $O(1)$
- `Remove-Max`: Remove the maximal element in time $O(\log n)$
- `Insert`: Insert a new element in time $O(\log n)$

While we often think of heaps as a tree, note that they can also be stored in memory as an array.

4.2 Disjoint-set Data Structure

As we've seen in Kruskal's algorithm for finding MST's, disjoint sets are useful data structures for maintaining sets that contain distinct elements. In this class, we represent them as trees, where the root of the tree is the canonical "name" of the tree.

The disjoint-set data structure enables us to efficiently perform operations such as placing elements into sets, querying whether two elements are in the same set, and merging two sets together. To make it work, we must implement the following operations:

1. `MAKESET`(x) – create a new set containing the single element x.
2. `UNION`(x, y) – replace sets containing x and y by their union.
3. `FIND`(x) – return the name of the set containing x.

We add for convenience the function `LINK`(x, y) where x,y are roots: `LINK` changes the parent pointer of one of the roots to be the other root. In particular, `UNION`(x, y) = `LINK`(`FIND`(x), `FIND`(y)), so the main problem is to make the `FIND` operations efficient.

Union By Rank and Path Compression To ensure $O(\log n)$ runtime of `FIND`, we try to ensure that the underlying tree structure is balanced under `UNION` operations. We do so by the union-by-rank heuristic which tries to preserve as low rank as possible by making the set with larger rank the parent. We also have path compression, which is the idea that if we ever do a find on a node, we should update its parent to directly point at the root.

4.3 LCA and RMQ

Least Common Ancestor: given two nodes in a tree, what is their last common ancestor?

Range Minimum Query: for an array of numbers, what is the index of the smallest number in a given subarray?

We can solve both using linear processing time, linear memory, and constant query time.

1. Linear-time reduction from LCA to RMQ:
   (a) DFS array V: an array of size $2n - 1$ that keeps track of nodes visited by DFS.
   (b) Level array L: same length as V, keeps track of distance from root at each node.
   (c) Representative array R: $R[i]$ is the first index of V containing the value i.
   (d) LCA(u, v) is now same as RMQ$(R[u], R[v])$ on L.
2. The reduction above guarantees that adjacent elements of L differ by no more than one (ï£¡1 RMQ). We can take advantage of this to get a linear-time preprocessing/memory algorithm for LCA and RMQ.
3. It is also possible to reduce general RMQ to ± 1 RMQ in linear time.
4. Consequently, we can solve these in $O(n)$ space and preprocessing time, and $O(1)$ query time.

5 Algorithmic Toolkit

5.1 Greedy

Greedy algorithms involve, broadly speaking, searching the space of solutions by only looking at the local neighborhood and picking the 'best' choice, for some metric of best.

This is a very useful paradigm for producing simple, fast algorithms that are even correct from time to time, even though they feel a lot like heuristics. It is surprising how well greedy algorithms can do sometimes, so the greedy paradigm is a reasonable thing to try when you're faced with a problem for which you don't know what approach might work (e.g., like on a final exam).

5.2 Divide and Conquer

The divide and conquer paradigm is another natural approach to algorithmic problems, involving three steps:

1. Divide the problem into smaller instances of the same problem;
2. Conquer the small problems by solving recursively (or when the problems are small enough, solving using some other method);
3. Combine the solutions to obtain a solution to the original problem.

Examples we've seen in class include binary search, matrix multiplication, fast integer multiplication, and median finding. Because of the recursion plus some additional work to combine, the runtime analysis of divide and conquer algorithms is often easy if we use the Master theorem.

5.3 Dynamic Programming

5.3.1 Approach

1. Define your subproblem clearly. Specify what exactly the inputs are and how they relate to your output.
2. Define your recurrence relation and base cases. Explain why your recurrence is correct.
3. Analyze the asymptotic runtime and memory usage.

5.4 Linear Programming

5.4.1 Review

1. *Simplex Algorithm*: The geometric interpretation is that the set of constraints is represented as a polytope and the algorithm starts from a vertex then repeatedly looks for a vertex that is adjacent and has better objective value (it's a hill-climbing algorithm). Variations of the simplex algorithm are *not* polynomial, but performs well in practice.
2. Standard form required by the simplex algorithm: *minimization, nonnegative variables and equality constraints.*
3. Getting the *dual* of a maximization problem:
   (a) Change all inequality constraints into $\leq$ constraints, negating both sides of an equation if necessary;
   (b) transpose the coefficient matrix;
   (c) invert maximization to minimization;
   (d) interchange the roles of the right-hand side and the objective function;
   (e) introduce a nonnegative variable for each inequality, and an unrestricted one for each equality;
   (f) for each nonnegative variable introduce a $\geq$ constraint, and for each unrestricted variable introduce an equality constraint.

6 Algorithmic Problems

6.1 Graph Traversal

Graph traversals are to graphs as sorting algorithms are to arrays – they give us an efficient way to put some notion of order on a graph that makes reasoning about it (and often, making efficient algorithms about it) or working on it much easier.

We've seen two major types of graph traversals:

- *Depth-First Search (DFS)*: keep moving along edges of the graph until the only vertices you can reach have already been visited, and then start backtracking. This also leads to a topological sort of a directed acyclic graph, which orders the vertices by an ancestor descendant relation. Runtime: $O(|V| + |E|)$.
- *Breadth-First Search (BFS)*: visit the vertices in the order of how close they are to the starting vertex. Runtime: $O(|V| + |E|)$ if all the edges have length 1, $O(|E| log |V|)$ for Djikstra's algorithm on nonnegative edge lengths with a binary heap.

Remember: the basic distinction between the two graph traversal paradigms is that DFS keeps a *stack* of the vertices (you can think of it as starting with a single vertex, adding stuff to the right, and removing stuff from the right as well), while BFS keeps a *queue* (which starts with a single vertex, adds stuff to the right, but removes stuff from the left) or a heap (which inserts vertices with a priority, and then removes the lowest priority vertex each time).

6.2 Shortest Paths

The shortest paths problem and various variants are very natural questions to ask about graphs. We've seen several shortest paths algorithms in class:

1. *Dijkstra's algorithm*, which finds all single-source shortest paths in a directed graph with non-negative edge lengths, is an extension of the idea of a breadth-first search, where we are more careful about the way time flows. Whereas in the unweighted case all edges can be thought of taking unit time to travel, in the weighted case, this is not true any more. This necessitates that we keep a *priority queue* instead of just a queue.
2. A single-source shortest paths algorithm (*Bellman-Ford*) that we saw for general (possibly negative) lengths consists of running a very reasonable local update procedure enough times that it propagates globally and gives us the right answer. Recall that this is also useful for detecting negative cycles!
3. A somewhat surprising application of dynamic programming (*Floyd-Warshall*) that finds the shortest paths between all pairs of vertices in a graph with non-negative weights by using subproblems $D_k[i, j]$ = shortest path between i and j using intermediate nodes among $1, 2, \dots, k$.

6.3 Minimum Spanning Trees

A tree is an undirected graph $T = (V, E)$ satisfying all of the following conditions:

1. T is connected,
2. T is acyclic,
3. $|E| = |V| - 1$.

However, any two conditions imply the third.

A *spanning tree* of an undirected graph $G = (V, E)$ is a subgraph which is a tree and which connects all the vertices. (If G is not connected, G has no spanning trees.)

A *minimum spanning tree* is a spanning tree whose total sum of edge costs is a minimum.

6.4 Network Flows

6.4.1 Review

1. The way that simplex algorithm works for Max-Flow problem: it finds a path from S to T (say, by DFS or BFS) in the *residual graph* and moves flow along this path of total value equal to the minimum capacity of an edge on the path.
2. A *cut* is a set of nodes containing S but not T. The *capacity* of a cut is the sum of the capacities of the edges going out of this set.
   - *Maximum flow* is equal to *minimum cut*.
3. **Ford-Fulkerson algorithm** (augmenting path algorithm): Another way to find the maximum flow of a graph with integer capacities is repeatedly use DFS to find an *augmenting path* from start to end node in the residual network (note that you can go backwards across edges with negative residual capacity), and then add that path to the flows we have found. Stop when DFS is no longer able to find such a path.
   - To form the residual network, we consider all edges $e = (u, v)$ in the graph as well as the reverse edges $\overline{e} = (v, u)$.
     (a) Any edge that is not part of the original graph is given capacity 0.
     (b) If $f(e)$ denotes the flow going across e, we set $f(\overline{e}) = -f(e)$.
     (c) The *residual capacity* of an edge is $c(e) - f(e)$
   - The runtime is $O((m + n)(\textbf{max flow}))$.

6.4.2 Exercises

6.5 2-player Games

An equilibrium in a 2-player game occurs when neither player would change his strategy even knowing the other player's strategy. This leads to two types of strategies:

- Pure strategy: a player always chooses the same option
- Mixed strategy: both players randomly choose an option using some probability distribution

For a player, option A is *dominated* by option B if, no matter what the other play chooses, the first player would prefer B to A. Because players will never play dominated strategies, the first thing to do when solving a 2-player game by hand is to eliminate any that you find.

7 Probabilistic Algorithms and Problems

7.1 Document Similarity

7.1.1 Motivation

Suppose we have many documents that we wish to compare for similarity in an efficient manner. We may have a search engine, and we don't want to display duplicate results. However, duplicate results (e.g., someone copied an article from another place) aren't exact duplicates, but rather exhibit high *s*imilarity. It's also okay if we don't have 100% accuracy.

7.1.2 Set Resemblance Problem

First, we discuss a problem that we can reduce document similarity to, as you will see in a moment.

1. Suppose that we have two sets of numbers A and B. Then their resemblance is defined as

$$\text{resemblance}(A, B) = R(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

2. Finding the exact resemblance is $O(n^2)$ if we compare the elements pairwise, and $O(n \log n)$ if we sort first and then compare. We wish to find an approximation of the resemblance in a much more efficient manner.

3. Suppose that we have functions $\{\pi_1, \pi_2, \pi_3, \dots\}$, where each π_k is a different random permutation from the 64-bit numbers to the 64-bit numbers (or whatever the space of numbers you are using). We define $\pi_k(A) = \{\pi_k(a) : a \in A\}$. Then we find that

$$\Pr[\min\{\pi_k(A)\} = \min\{\pi_k(B)\}] = \frac{|A \cap B|}{|A \cup B|} = R(A, B)$$

   This is because every element in $A \cup B$ has an equal probability of mapping to the minimum element after the permutation is applied.

4. Thus, the more random permutations we use, the more confidence we have about the resemblance of A and B.

7.1.3 Document Similarity Solved

1. In order to turn the document similarity problem into the set resemblance problem, we must first transform a document into a set of numbers. We do this by the process of *s*hingling, where we hash overlapping pieces of the document (e.g., every four consecutive words), which gives us a set of numbers. Let this set be called S_D.

2. Now for every document, all we need to do is store is a *s*ketch of the document, which would be an ordered list of numbers $(\min\{\pi_1(S_D)\}, \min\{\pi_2(S_D)\}, \min\{\pi_3(S_D)\}, \dots, \min\{\pi_{100}(S_D)\})$. The larger the sketch, the more confidence you'll have about the resemblance, but the more memory and time it'll take to process each document.

3. Now to compare the similarity of two documents, we just need to compare their sketches. We'll set a threshold for the number of matches that need to occur in order for two documents to be considered similar. For example, we can say that two documents whose sketches match at 90 out of 100 numbers are considered similar. As you can see, this process of comparing the similarity of two documents is quite efficient!

4. The probability $p(r)$ that at least 90 out of 100 entries in the sketch match stays very low until r approaches 0.9 and then grows very quickly to 1. This property of the function means that we won't make mistakes too often!

7.2 Primality Testing

7.2.1 Review

1. *Fermat's Little Theorem*: For all primes p, $a^{p-1} = 1 \mod p$ for all integers a that are not multiples of p.

2. A composite number n is said to be a-pseudoprime if $\gcd(a, n) = 1$ and $a^{n-1} = 1 \mod n$. *Carmichael numbers* are composite numbers n such that n is a-pseudoprime for all a that do not share a factor with n.

3. *Miller-Rabin Primality Test*: Suppose we have an odd number n, where $n - 1 = 2^t u$, and we want to check if it is prime. Then we proceed as follows:

   (a) Randomly generate a positive integer $a < n$.

   (b) Calculate $a^u, a^{2u}, a^{4u}, \ldots, a^{2^t u} = a^{n-1} \mod n$.

   (c) Check if $a^{n-1} = 1 \mod n$. If it is not, n is composite. If it is, let $x = a^{n-1} = a^{2^t u}$ (already calculated).

   (d) There are three possibilities for x:

      i. $x = 1$: then if $x = a^u$ end, otherwise let $x = x^{1/2}$ and repeat 4.

      ii. $x = -1$: then the algorithm is indecisive and ends.

      iii. $x \notin \{-1, 1\}$: then n must be composite and a is a witness

   (e) Repeat the above steps several times until we get bored or n is determined to be composite. If n is not found to be composite, then return that it is (probably) prime.

   It turns out that for any composite number n, the probability that a randomly chosen a will reveal it as composite (i.e., be a witness to the compositeness of n) is $3/4$. So by iterating enough times, the probability of error can be reduced below the probability that your computer crashes during the execution.

7.2.2 Exercises

7.3 Cryptography

7.3.1 Review

1. RSA is an encryption algorithm loosely based on the assumption that factoring is difficult. In order to efficiently implement RSA, many of the algorithms previously discussed in this course (Extended Euclidean algorithm, repeated squaring, Miller-Rabin testing) are used.

    - Set-up: Alice uses Miller-Rabin to find a prime numbers p, q and sets $n = pq$. She then chooses $e < n$ and publishes a public key (e, n), and she privately calculates a secret key d such that $ed = 1 \mod (p-1)(q-1)$.
    - Encryption: $E(x, n, e) = x^e \mod n$
    - Decryption: $D(c, d) = c^d \bmod n = x^{ed \mod (p-1)(q-1)} \mod n = x$

7.4 Hashing

A *hash function* is a mapping $h : \{0, \ldots, n-1\} \to \{0, \ldots, m-1\}$. Typically, $m << n$.

Hashing-based data structures (e.g. *hash tables*) are useful since they ideally allow for constant time operations (lookup, adding, and deletion). However, the major problem preventing this is collisions, defined as when we hash an item to a location that already contains an item. Collisions occur more, if m is too small or if a poorly chosen hash function is used..

In reality, it is hard to get perfect randomness, where all elements hash independently.

7.5 Random Walks

A *random walk* is an is an iterative process on a set of vertices V. In each step, you move from the current vertex v_0 to each $v \in V$ with some probability. The simplest version of a random walk is a one-dimensional random walk in which the vertices are the integers, you start at 0, and at each step you either move up one (with probability 1/2) or down one (with probability 1/2).

2-SAT: In lecture, we gave the following randomized algorithm for solving 2-SAT. Start with some truth assignment, say by setting all the variables to false. Find some clause that is not yet satisfied. Randomly choose one of the variables in that clause, say by flipping a coin, and change its value. Continue this process, until either all clauses are satisfied or you get tired of flipping coins. We used a random walk with a completely reflecting boundary at 0 to model our randomized solution to 2-SAT. Fix some solution S and keep track of the number of variables k consistent with the solution S. In each step, we either increase or decrease k by one. Using this model, we showed during section that the expected running time of our algorithm is $O(n^2)$.

8 Solving “Hard" Problems

This section primarily deals with NP-hard problems and methods of dealing with them, not just problems difficult to students.

8.1 NP-completeness

8.1.1 Review

1. P is the class of all yes/no problems which can be solved in time which is polynomial in n, the size of the input.
2. P is closed under polynomial-time reductions:
   (a) A **reduction** R from Problem A to Problem B is an algorithm which takes an input x for A and transforms it into an input $R(x)$ for B, such that the answer to B with input $R(x)$ is the answer to A with input x.
   (b) The algorithm for A is just the algorithm for B *composed* with the reduction R.
   (c) *When doing a reduction, remember to check that it's going in the right direction!*

   Problem A is *at least as hard as* Problem B if B reduces to it (in polynomial time): If we can solve A, then we can solve B. We write:

   $$A \geq_{\mathrm{P}} B \quad \text{or simply} \quad A \geq P. \tag{1}$$

3. NP is the class of yes/no problems such that if the answer if yes, then there is a **short** (polynomial-length) **certificate** that can be checked in polynomial time to prove that the answer is correct.

   Examples: Compositeness, 3-SAT are in NP.

   Not-satisfiable-3-SAT is not in NP.

   The complement of an NP problem may not be in NP: e.g. if P $\neq$ NP, then Not-satisfiable-3-SAT is not in NP.

4. **NP-complete**: In NP, and all other problems in NP reduce to it. That is, A is NP-complete if

   $$A \in \mathrm{NP} \quad \text{and} \quad A \geq_{\mathrm{P}} B \ \forall B \in \mathrm{NP}.$$

   NP-hard: All problems in NP reduce to it, but it is not necessarily in NP.

5. NP-complete problems from lecture: circuit SAT, 3-SAT, integer LP, independent set, vertex cover, clique.
6. Remember: while we strongly believe so, we don't know whether P $\neq$ NP!

8.2 Local search

8.2.1 The basic idea

1. Can be thought of as “hill climbing."
2. Importance of setting up the problem: Need to define the notion of **neighbourhood** so that the state space is nice, and doesn't have lots of local optima.
3. Choosing a starting point is important. Also, how we move to neighbours can be important | e.g. whether we always move to the best neighbour or just choose one that does better, and also how we choose to break ties.

8.2.2 Variations

1. Metropolis rule: random neighbours, with higher likelihood of moving to better neighbours.
2. Simulated annealing: Metropolis with decreasing randomness (cooling) over time.
3. Tabu search: adds memory to hill climbing to prevent cycling.
4. Parallel search, genetic search.

8.3 Approximation algorithms

8.3.1 Review

1. **Vertex cover**: Want to find minimal subset $S \subseteq V$ such that every $e \in E$ has at least one endpoint in S.

   To do this, repeatedly choose an edge, throw both endpoints in the cover, delete endpoints and all adjacent edges from graph, continue. This gives a 2-approximation.
2. **Max Cut**: see homework problem.
   (a) Randomized algorithm: Assign each vertex to a random set. This gives a 2-approximation in expectation.
   (b) Deterministic algorithm: Start with some fixed assignment. As long as it is possible to improve the cut by moving some vertex to a different set, do so. This gives a 2-approximation.
3. **Euclidean traveling salesman**: Find MST, create “pseudo-tour,” take shortcuts to get tour. This gives a 2-approximation.
4. **MAX-SAT**: Asks for the maximum number of clauses which can be satisfied by any assignment. LP relaxation and randomized rounding.