CS6515 Exam 3 Notes

Linear Programming

Primal Form (Standard Form)
max c^t x
Ax <= b
X >= 0

Dual Form
Min y^Tb
y^T A >= c^T
y >= 0

Weak Duality Theorem
- the solution to the primal problem is always less than or equal to the solution to the dual problem (the duality gap >= 0)
- Boundness only makes sense if the LP is feasible

Prime/Dual	Dual/Prime
Unbounded ->	Infeasible
Infeasible ->	Unbounded OR Infeasible
Bounded ->	Bounded AND Feasible

Strong Duality Theorem
- If an LP has a bounded optimum, then so does it dual
- Primal form and dual form do not have the same min or max points, but the same objective value

Max Flow Problem as LP
- Variables are flows on each edge
- Constraints are conservation and flow capacity
- Max source outflow or sink inflow or add a flow back link

NP-Complete

Class Definitions

P: The problem can be solved and verified in polynomial time
NP: Problems that can be verified in polynomial time, or solvable by nondeterministic Turing machine in polynomial time
NP-Complete: the problem is in both NP and NP-Hard
NP-Hard: As hard as the hardest problems in NP, contains NP complete
Generalization: A is a special case of B. if B is a generalization of A, such that A is a subset of the instances of B. There is a reduction from A to B in which both the f and h transforms are the trivial transformation.

Pseudo-polynomial: Polynomial to the magnitude of the input, but exponential to the size of the input. Ex: knapsack, max flow

Reduction Steps

1. Show B is NP (complexity of solution verification)
2. Input transformation: I -> f(I) (complexity of transformation)
3. Solution transformation: S -> h(S) (complexity of transformation)
4. Bidirectional proof that a solution for B exists IFF a solution for A exists:
    1. If a solution for B exists, then a solution for A exists
    2. If no solution for A exists, then no solution for B exists

Known Problems

- **SAT**
    - Input: boolean formula f in CNF with n variables and m clauses
    - Output: Satisfying assignment S where each clause evaluates to TRUE. Returns 'NO' when no such S exists.
- ***SAT Variations***
    - k-SAT is SAT with an additional input constraint k so that each clause has at most k literals. To be NP-C, k > 2
    - 3-SAT restricted to no variable appears in more than three clauses (no literals more than twice) is still NP-C (3-at-most-3SAT)
- **Knapsack (Search)**
    - Input: n-items with integer weights: w_1,...,w_n; integer values v_1,...,v+n; capacity B; and goal g
    - Output: subset S items where the total value V is >=g and total weight W is <= B. Returns 'NO' when no such S exists.
- **Subset Sum**
    - Input: n positive integers a_1, ..., a_n and a target, t
    - Output: Subset S of [1, ..., n] where the sum of a_i for all i in S = t, if such a subset exists. Returns 'NO' when no such S exists.
- **Independent-Set (Search)**
    - Input: Graph G = (V, E) and goal g
    - Output: Subset S in V if there are **no edges** between every vertex in S and |S|>=g. Returns 'NO' when no such S exists.
- **Clique (*Search*)**
    - Input: Undirected Graph G = (V, E) *and goal g*
    - Output: subset S in V where there **are edges** between every pair of vertices in S and |S|>=g. Returns 'NO' when no such S exists.
- **Vertex Cover (Search)**
    - Input: Undirected Graph G = (V, E) and budget, b
    - Output: Subset S in V, if every e=(x, y) has either x in S or y in S and |S|<=b. Returns 'NO' when no such S exists.
- **Set Covering**

    - Given a set of elements {1, 2, ..., *n*} (called the universe) and a collection *S* of *m* subsets whose union equals the universe, the set cover problem is to identify the smallest sub-collection of *S* whose union equals the universe
    - Input: A set of elements B; sets S1,...,Sm ⊆ B
    - A selection of the Si whose union is B.
- **Hitting Set**
    - Input: A set, A = {a_1...a_n}, and a collection of sets, {S_1,... S_n}, and k
    - Output: H in A such that H intersect S_i is non empty for any I, and |H| <= k
- **K-Colorings**
    - Input: Undirected Graph G=(V,E) & integer k >2
    - Output: assign each vertex a color in {1, 2, ..., k} so that adjacent vertices get different colors and NO if no such k-coloring exists
    - *Note:* for k=2, K-Colorings∈P because it can be solved with modified BFS/DFS to find a bipartite graph
- **Traveling Salesman**
    - Input:
        - n vertices {1, 2, ..., n}
        - n(n-1)/2 pairwise distances between each vertex
        - A budget b for total distance traveled
    - Output: a path that visits every vertex with a total distance <= b. Returns 'NO' when no such path exists.
- **Rudrata Cycle**
    - Input: G = (V, E) (directed or undirected)
    - Output: A cycle C that visits every vertex exactly once. Returns 'NO' when no such C exists.
- **Rudrata Path**
    - Input: G = (V, E), s, t. Starting at s and ending at t.
    - Output: Returning a path P that visits each vertex exactly once. Returns 'NO' when no such such P exists
- **Balanced Cut**
    - Input: G = (V, E) and a budget b,
    - Output: partition the vertices into two sets S and T such that |S|,|T| >= n/3 and such that there are at most b edges between S and T. Returns 'NO' when no such S exists.
- **Zero One Equations**
    - Input: m x n matrix A, where entries are 0 or 1
    - Output: find a n-vector x, such that Ax = 1, where 1 denotes the all 1 vector

Reduction Gadgets

CNF Manipulations

- Reduction of clause length:
  - (a1 ∨ a2 ∨ · · · ∨ ak) -> (a1 ∨a2 ∨y1) (!y1 ∨a3 ∨y2) (!y2 ∨a4 ∨y3) · · · (!yk−3 ∨ak−1 ∨ak),
- Reduction of variable occurrence:
  - For variables that occur more than n times, replace each occurrence by x_1,...,x_n
  - Add new clauses in the format to force the auxiliary variables to have the same value:
    - (x_1! || x_2) && (x_2! || x_3) ... && (x_n! || x_1)
- Padding of clause length:
  - Add an auxiliary variable for each missing literal, and all possible combinations of all the auxiliary variables
    - (x_1 || !y) && (x_2 || y)
- Examples: SAT to 3SAT, 3SAT to 3-at-most-SAT, 3-SAT to 4-SAT

Graph Properties and Manipulations

- **k**-clique has k(k-1)/2 edges
- G complement = (V, !E), containing ordered pairs of vertices in V not in E
- VC to Clique: Given G = (V, E), a subset C in V is a vertex cover in G if and only if G - C is a clique in G complement = (V, V^2 -E)
- IS to Clique: S is an independent set in G = (V, E) , then it's a clique in G complement = (V, !E)
- VC to IS: S is a vertex cover of G = (V, E) iff V - S is an independent set of G
- Add or remove components of the graph to form the needed structures like clique or IS
  - These elements must be excluded from the solution
- Feed a formed IS or clique into the input
- Examples: IS to Vertex Cover, IS to Clique, Vertex Cover to Clique, IS to Star Graph, IS to Clique and IS, Rudrata Path to Rudrata Cycle

Set Covering Reductions

- Set Cover to Vertex Cover
  - Each element is an edge, each set is a vertex connected to the edges in the subset
- Set Cover to Hitting Set
  - Maps n numbers to n empty sets, then m sets to m numbers, and add the numbers into the sets which are in the SC sets
- Vertex Cover to Hitting Set

- Map edges to the sets, vertexes to the elements in the sets. e= (u, v) -> s = {u, v}
- the elements picked are the vertices for the cover

SAT to Graph Reductions

- 3-SAT to IS:
  - form a triangle or bar for each clause where each vertex is a literal
  - Add edges between literals with the same variables
  - Input to IS the graph and m, so IS has to pick one vertex from each subgraph
- 3-SAT to K-coloring:
  - Create three colors, T, F and base
  - Create a shape with one base vertex and a pair of T and F vertex for each variable in hub and spoke model
  - Create an OR gadget for every clause in the form: or two or gates
  - Connect all OR gadgets to the hub and spoke with one edge to the base and one edge to the F node of each variable, so each OR gadget is forced to be true
- 3-SAT to 3D matching:
  - For each variable x_i, there is a "variable gadget" shaped like a wheel. It is made of overlapping triplets. The number of triplets is twice the number of occurrences of x_i in the formula. There are exactly two ways to cover all the vertices in the gadget: one is to choose all even-indexed triplets, and one is to choose all odd-indexed triplets. These two ways correspond to setting x_i to "true" or "false". The "true" selection leaves uncovered exactly one vertex in every odd-indexed triplet, and the "false" selection leaves uncovered exactly one vertex in every even-indexed triplet.
  - For each clause x_i u x_j u x_k, there is a "clause gadget" shaped like a rose. It is made of three overlapping triplets, one for each variable in the clause. It can be covered iff at least one of the nodes is left uncovered by the selection of the variable gadgets.
  - Since it is possible that two or more nodes are left uncovered, we also need a "garbage collection gadget". It is shaped like a larger rose. It is made of several overlapping triplets, one for each vertex that can be left uncovered in the variable gadget. The number of such gadgets is determined so that they can be covered exactly if and only if there is a satisfying assignment.

Summation Reductions

- 3 Sat to Subset Sum
  - Construct a subset sum problem, where the inputs are (n + m) digits long
  - Create two inputs for each variable, and two inputs for each clauses, and a target t
  - The number is constructed as n variables digits + m clause digits
  - Put 1 in the variable digit of the two input variables and t
  - Put 1 in the clause digit of the corresponding literals and 3 in t
  - Put 1 in the two buffer rows for each clause

    - Then the rows define the numbers for input to Subset Sum
- ZOE to Subset Sum
    - Interpret columns of matrix A as binary numbers, and the vector x is a linear combination to get the result vector B
    - Do the summation in a base that doesn’t carry
- 3D Matching to ZOE
    - Express each 3D matching triple as a column in A, find a linear combination so that it sums up to the 1 vector
- Subset Sum to Knapsack
    - Wi = Si, Vi = Is, then capacity = k, value goal = k