15.053 Exam 3 Notes

Minimum Cost Flow

- General formulation of problem
    - Specified net demand b_i for every node. >0 for demand node, <0 for supply node, $=0$ for transhipment node.
    - Cost $c_{i,j}$ and capacity $u_{i,j}$ for every arc.
    - Objective – minimize $\sum_{\text{all arcs}} c_{i,j} x_{i,j} = \boldsymbol{c} \cdot \boldsymbol{x}$
    - Conservation constraints $\sum_{\text{all arcs in}} x_{i,k} - \sum_{\text{all arcs out}} x_{k,j} = b_k \Rightarrow \boldsymbol{Ax} = \boldsymbol{b}$
    - Non-negativity and capacity constraints for each arc $0 \le x_{i,j} \le u_{i,j} \Rightarrow 0 \le \boldsymbol{x} \le \boldsymbol{u}$
- Dummy indices
    - If $\sum b_k > 0$, the problem is infeasible – demand exceeds supply.
    - If $\sum b_k < 0$, need to add a dummy index to consume excess demand via zero-cost arcs from all source/supplies.
- The matrix $\boldsymbol{A}$
    - Items are either +1 and –1.
    - **Column → arc** *and* **row → node conservation constraint**.
    - Obviously, each column has exactly one "+1" and one "–1".
- Notes
    - Any LP method will work to solve this
    - Network simplex method works better for minimum cost flow (MCF) problem.
    - If supplies, demands and capacities are integer, the optimal solution will be integer.
- Formulating shortest path as MCF problem
    - Start node capacity –1, final node capacity +1, other capacity 0, arcs have capacity 1.
    - Integrality of solutions implies that each arc *will* have flow 0 or 1.
    - Thus, can solve shortest path as LP.

Other problems

- **Maximum flow problem**
  - General formulation of problem
    - Source and sink node, and capacity for each arc (no costs).
  - Dealing with it as MCP
    - Insert **return arc** from t to s to capture max flow. Cost –1, infinite capacity.
    - All other arcs are given cost 0. All nodes are transhipment.
- **Transportation problem**
  - General formulation of problem
    - Set of factories N_1 with supply s_i and of markets N_2 with demand d_j
    - $c_{i,j}$ cost of shipping from factory i to market j.
    - $\sum s_i = \sum d_j$
- **Choosing bids optimally**
  - General formulation of problem
    - Group N_1 and N_2
    - Set of allowable assignments with costs $c_{i,j}$
  - Formulation as MCP
    - Each assignment becomes a directed edge of infinite capacity.
    - Each in N_1 becomes a source of 1, each in N_2 becomes a sink of 1
    - Require flow to minimise cost
    - If $N_1 > N_2$, add dummies with cost 0 linked to *all* the N_1.
- **Including time effects**
  - Just replicate every node for every time period – flows can be between nodes of different times or same times.
- **Multi-commodity flow problem**
  - General formulation
    - Set of commodities Q

- $c_{q,\, i,\, j}$ and $u_{q,\, i,\, j}$ – unit cost and capacity of flow of commodity Q on arc $(i,\, j)$
- $u_{i,\, j}$ – shared capacity of arc $(i,\, j)$.
- $b_{q,\, k}$ – net demand of commodity q at node k.
- Decision is $x_{q,\, i,\, j}$ – net flow of commodity q on arc $(i,\, j)$.

- Program
  - Minimise $\sum_q \sum_{(i,j)} c_{q,i,j} x_{q,i,j}$
  - Flow constraint for each node and each commodity
$$\sum_{\text{flow in}} x_{q,i,k} - \sum_{\text{flow out}} x_{q,k,j} = b_{q,k}$$
  - Total capacity constraint at each arc
$$\sum_Q x_{q,i,j} \le u_{i,j}$$
  - Non-negativity and flow capacity at each arc for each capacity
$$0 \le x_{q,i,j} \le u_{q,i,j}$$

Integral Linear Programming

- **Knapsack problem**
  - General formulation of problem
    - Items $\{1, \ldots, n\}$ to put into sack.
    - Each item has weight w_i and value c_i
    - Maximum weight knapsack can hold is b
    - Decision variables
$$x_i = \begin{cases} 1 & \text{choose item } i \\ 0 & \text{otherwise} \end{cases}$$
    - Maximise $\sum c_i x_i$ such that $\sum w_i x_i \le b$ and $x_i \in \{0,1\}$
  - Modifications
    - If E is a set of mutually exclusive choices, we require $\sum_{i \in E} x_i \le 1$.

        - If we want *at least* one element from E to be chosen (we want to *cover* E), we require $\sum_{i \in E} x_i \geq 1$.
        - If we want exactly some element from E to be chosen, we require $\sum_{i \in E} x_i = 1$.
        - If j can only be chosen if k is also chosen, require $x_j \leq x_k$

- **Fixed Charges**
    - Let's imagine we have a cost that is non-linear

$$\theta(x_j) = \begin{cases} f_j + h_j x_j & x_j > 0 \\ 0 & \text{otherwise} \end{cases}$$

    - We can deal with this non-linear cost by introducing a new binary variable. We require that $x_j \leq My_j$, with M a very large number, or the upper bound on x. This ensures that

$$y_j = \begin{cases} 1 & x_j > 0 \\ 0 & \text{otherwise} \end{cases}$$

    - The objective cost then becomes $f_j y_j + h_j x_j$.
- **Matchings**
    - General formulation of problem
        - A **matching** is a set of **edges** such that no two edges share a common node.
        - Decision variables

$$x_{\{i,j\}} = \begin{cases} 1 & \text{if edge } \{i,j\} \text{ is on the matching} \\ 0 & \text{otherwise} \end{cases}$$

        - For each node, ensure only one adjacent edge selected

$$\sum_{\{i,k\}} x_{\{i,k\}} \leq 1$$

        [This applies for each node i]
        - Objective function is max $\sum_{\{i,j\}} w_{\{i,j\}} x_{\{i,j\}}$, where w is the weight of a given pairing, if any.
- **Graph colouring**
    - An assignment of colours to nodes such that no two adjacent nodes have the same colour. Need at most n colours if n nodes.
    - General formulation of problem

- $x_{i,\,c}$ is 1 if colour c is assignment to node I, 0 otherwise.
- y_c is 1 if colour c is used, 0 otherwise.
- Objective is to minimise $\sum_{c \in K} y_c$
- Every node is assigned only one colour, so $\sum_i x_{i,c} = 1$
- Each colour can be assigned to an edge at most once, so

$$x_{i,c} + x_{j,c} \le 1 \qquad \text{for all } \{i,j\} \text{ and } c$$

- Require

$$x_{i,c} \le y_c \qquad \text{each } i \text{ and } c$$

- In general, any graph arising from a planar map can be coloured with four colours.

- **<u>Travelling salesman problem</u>**
  - Decision variables $x_{\{i,j\}} = 1$ if node is on tour, 0 otherwise.
  - Objective function is simple; minimise $\sum c_{\{i,j\}} x_{\{i,j\}}$.
  - Require it to be a tour – so $\sum_{\{i,k\}} x_{\{i,k\}} = 2$ for each node i. This ensures that a single edge enters and leaves each node.
  - To ensure that there are no subtours, we require $\sum_{\{i,j\} \in \delta(S)} x_{\{i,j\}} \ge 2$ where S is any proper subset of nodes, and $\delta\{S\}$ is the set of all edges with one point in S and one point outside S.

Relaxation

- An IP P can be relaxed to an LP P' by allowing integer variables to become continuous (though with the same range).
- The relaxed model must have an equal or better than optimal solution than the unrelaxed model.
- If the relaxed model is unfeasible, then so is the unrelaxed model.
- If the optimal solution of P' is also feasible in P, then it is the optimal solution of P.
- We might be able to round the optimal solution of P' and get a feasible solution of P, which gives us a <u>bound</u> for that linear program.

- A **valid inequality** for a given program P holds for all **integer** feasible solutions to P. To strengthen a relaxation, a valid inequality must cut off some feasible solution to the current LP relaxation that are not valid in the true ILP model.
- If we add such a valid inequality, we get a **stronger relaxation**, and an **optimal value** that is **closer** to the **true one**.
- In short – solution to P' gives upper bound on solution, rounding solution to P' gives lower bound on solution.

Branch & Bound

- Create **enumeration tree one branch and one node at a time.**
- Before branching, solve **relaxation of candidate LP at that node**
- Could **terminate** because of
  - **Infeasibility**
  - **Bound**: best possible solution (solution of relaxation) is less than **incumbent solution**
  - **Solving**: if the solution of the relaxed LP is integer, stop there. If it's better than the incumbent, replace the incumbent. Otherwise, ignore it.
- Otherwise, we **branch**, changing the variable that is **non-integer** to change.
- When we discover a new incumbent solution, any active node with candidate optimal solution less than the new one can be eliminated.
- *Active nodes* have no children, and have not been terminated.
- Can take the highest parent bound on all active nodes to find the upper bound on the solution.
- **Depth first**: always do the deepest next. **Best first**: each iteration, start with the bets. **Depth forward best back**: do it normally, but when finishing an iteration, go back to the best.