Last topic: Summary; Heuristics and Approximation Algorithms

Topics we studied so far:

- Strength of formulations; improving formulations by adding valid inequalities
- Relaxations and dual problems; obtaining good bounds on the optimal value of an IP
- Cutting plane algorithms (Exact general purpose algorithm based on strengthening LP relaxations; does not produce a feasible solution until termination)
- Branch and bound algorithms (Exact general purpose algorithm using primal and dual bounds to intelligently enumerate feasible solutions; does not produce a feasible solution until one of the subproblems is pruned by optimality)

Solving IPs in real life: Special classes of problems

- Can we devise specialized algorithms for a particular class of problems?
- What if there is a need to get an OK feasible solution quickly?
A heuristic or approximation algorithm may be used when...

- A feasible solution is required rapidly; within a few seconds or minutes
- An instance is so large, it cannot be formulated as an IP or MIP of reasonable size
- In branch-and-bound algorithm, looking to find good/quick primal bounds on subproblems solutions sufficiently quickly

Provide a feasible solution without guarantee on its optimality, or even quality, or algorithm running time.

Well-designed heuristics for particular problem classes have been empirically shown to find good solutions fast.

Examples of heuristic algorithms:

- **Lagrangian** Solve the Lagrangian Relaxation with a good value of $u$. If the solution $x(u)$ is not feasible, make adjustments while keeping objective function deteriorations small.
- **Greedy** Construct a feasible solution from scratch, choosing at each step the decision bringing the “best” immediate reward.
- **Local Search** Start with a feasible solution $x$, and compare it with “neighboring” feasible solutions. If a better neighbor $y$ is found, move to $y$, and repeat the same procedure. If no such neighbor is found, the process stops — a local optimum is found.

In practice, often used in combination: a greedy procedure or other heuristic to construct a feasible solution, followed by a local search to improve it.
Examples of greedy heuristics

**Example 1: 0–1 Knapsack**

\[
\max \left\{ \sum_{j=1}^{n} c_j x_j : \sum_{j=1}^{n} a_j x_j \leq b, \ x \in B^n \right\}, \ \text{where } a_j, c_j > 0 \ \forall j
\]

Assume the items are ordered so that \( \frac{c_j}{a_j} \geq \frac{c_{j+1}}{a_{j+1}} \), \( j = 1, \ldots, n-1 \)

The algorithm:

- **Initialization**: Start with no items in the knapsack
- **For** \( j = 1, \ldots, n \): if the space remaining in the knapsack is sufficient to accommodate \( j \)th item, include it; otherwise, skip it.

For example, if \( c = (12, 8, 17, 11, 6, 2, 2) \), \( a = (4, 3, 7, 5, 3, 2, 3) \), and \( b = 9 \), the solution obtained by the greedy heuristic is \( x^G = (1, 1, 0, 0, 0, 1, 0) \) with \( z^G = c^T x^G = 22 \) (the optimal solution is \( x_1 = x_4 = 1 \) with \( z = 23 \)).

**Example 2: STSP** \( G = (V, E), c_{ij} = c_{ji} \)

Greedy algorithm I:

- **Initialization**: Let \( S = \{1\} \), \( N = V \setminus S \), \( v_1 = 1 \)
- **For** \( j = 2, \ldots, |V| \), find \( v_j = \arg\min_{i \in N} c_{v_{j-1}, i} \). Set \( S = S \cup \{v_j\} \), \( N = N \setminus \{v_j\} \)
- Connect \( v_n \) to \( v_1 \). Resulting tour: \( v_1, v_2, \ldots, v_n, v_1 \)

Greedy algorithm II:

- **Initialization**: Let \( S = \emptyset \), \( N = E \). Sort the edges so that \( c_{e_k} \leq c_{e_{k+1}} \) for \( k = 1, \ldots, |E| - 1 \). Set \( j = 0 \).
- **Repeat until** \( S \) is a tour: increment \( j \); If \( S \cup \{e_j\} \) contains no subtours and no nodes of degree more than 2, set \( S = S \cup \{e_j\} \)
Examples of greedy heuristics

**Example 3: Minimum Spanning Tree problem** Given a connected graph $G = (V, E)$ with $c_e$, $e \in E$, construct a minimum-cost spanning tree, i.e., subgraph $\tilde{G} = (V, \tilde{E})$ which is connected and contains no cycles.

**Prim’s algorithm:**

- **Initialization:** Start with a graph $(V_1, E_1)$, where $|V_1| = 1$, and $E_1 = \emptyset$.
- For $k = 2, \ldots, |V|$, consider all edges $\{i, j\}$ with $i \in V_k$ and $j \notin V_k$, and choose the one with smallest cost. Let $V_{k+1} = V_k \cup \{j\}$ and $N_{k+1} = N_k \cup \{\{i, j\}\}$.

**Theorem** Prim’s algorithm finds an optimal solution of the MST problem.

**Proof** Relies on the cut optimality necessary and sufficient condition of MSTs.

Examples of local search

**Local Search Heuristics:** Start with a feasible solution $x$, and compare it with “neighboring” feasible solutions. If a better neighbor $y$ is found, move to $y$, and repeat the same procedure. If no such neighbor is found, the process stops — a local optimum is found. To define a local search, need a starting solution and a definition of a neighborhood.

**Example 1: local search heuristic for cost-minimizing UFL**

Observation: if a set of facilities $S \neq \emptyset$ is built, it is easy to find the cheapest feasible assignment of clients to these facilities. The total cost of such solution is $c(S) + \sum_{j \in S} f_j$, where $f_j = \sum_{i=1}^{m} \min_{j \in S} c_{ij}$.

**Local search:** Start with a set of depots $S \neq \emptyset$. Define a neighborhood of $S$ to be all non-empty sets that can be obtained by either adding or deleting one depot to/from $S$ (note: $S$ has at most $n$ neighbors).
Examples of local search

**Example 2: 2OPT heuristic for STSP** Given a set of edges $S$ that form a TSP tour, there is no other tour that differs from $S$ by exactly one edge. If two (disjoint) edges are removed from $S$, there is exactly one other tour that contains the remaining edges. In the 2OPT heuristic for STSP a neighborhood of $S$ consists of all tours that can be obtained removing two (disjoint) edges from $S$, and replacing them with two other edges. Note: $S$ has $n^2$ neighbors. Thus, at every iteration, the algorithm may have to compare up to $n^2$ tours.

Adapting local search for global optimization

Local search algorithms find a solution that is a local optimum, i.e., best in its neighborhood. Globally optimal solutions may not lie in that neighborhood!

The following are search algorithms designed to seek a globally optimal solution:
- Tabu search
- Simulated annealing
- Genetic algorithms
Approximation algorithms

Essentially, heuristics with a provable guarantee on the quality of the obtained solution.

**ε-Approximation Algorithm**

A heuristic algorithm for a class of problems constitutes an ε-approximation algorithm if for each instance of a problem in this class with optimal value $z$, the heuristic algorithm returns a feasible solution of value $z^H$ such that

$$z^H \leq (1 + \epsilon)z \text{ for a minimization problem},$$

or

$$z^H \geq (1 - \epsilon)z \text{ for a maximization problem}.$$

Additionally, keep an eye on the running time of the algorithm — make sure it grows at a reasonable rate as larger problems and/or smaller ε values are considered. (E.g., polynomial, not exponential, growth.)

**Examples of approximation algorithms: integer knapsack**

$$z = \max \left\{ \sum_{j=1}^{n} c_jx_j : \sum_{j=1}^{n} a_jx_j \leq b, \ x \in \mathbb{Z}_+^n \right\}, \text{ where } a_1, \ldots, a_n, b \in \mathbb{Z}_+.$$

Assume that $a_j \leq b \ \forall j$, and $\frac{c_i}{a_1} \geq \frac{c_i}{a_j} \ \forall j$.

The greedy solution $x^H$ has $x^H_1 = \left\lfloor \frac{b}{a_1} \right\rfloor$, with $z^H \geq c_1 \left\lfloor \frac{b}{a_1} \right\rfloor$.

**Theorem 12.1**

$$\frac{z^H}{z} \geq \frac{1}{2} \quad \text{(An } \varepsilon \text{-approximation with } \varepsilon = \frac{1}{2}).$$

**Proof**

The solution to LP relaxation $x^{LP}_1 = \frac{b}{a_1}$ and $z^{LP} = \frac{c_1b}{a_1} \geq z$.

Note: $\frac{b}{a_1} \leq 2\left\lfloor \frac{b}{a_1} \right\rfloor$ (since $\left\lfloor \frac{b}{a_1} \right\rfloor \geq 1$), so

$$\frac{z^H}{z} \geq \frac{z^H}{z^{LP}} \geq \frac{c_1 \left\lfloor \frac{b}{a_1} \right\rfloor}{c_1 \frac{b}{a_1}} \geq \frac{\left\lfloor \frac{b}{a_1} \right\rfloor}{2\left\lfloor \frac{b}{a_1} \right\rfloor} = \frac{1}{2}.$$
Examples of approximation algorithms: STSP with $\Delta \leq$

Consider an instance of STSP on a complete graph with
$0 \leq c_{ij} \leq c_{ik} + c_{jk} \ \forall i, j, k$

Tree Heuristic:
- Find a minimum spanning tree (let its length be $z_T \leq z$)
- Construct a “walk” that starts at node 1, visits all nodes and return to node 1 using only edges of the tree. The length of the walk is $2z_T$, since every edge is traversed twice.
- Convert the walk into a tour by skipping intermediate nodes that have already been visited. Due to triangle inequality, the length does not increase.

$z^H \leq 2z_T \leq 2z$.

An $\epsilon$-approximation with $\epsilon = 1$.
Can be improved to $\epsilon = 1/2$ by finding a minimum perfect matching on the nodes that have odd degree in the tree and using its arcs in the tour.

Examples of approximation algorithms: $0 - 1$ knapsack

$$z = \max \left\{ \sum_{j=1}^{n} c_jx_j : \sum_{j=1}^{n} a_jx_j \leq b, \ x \in B^n \right\}$$

Can solve in time proportional to $n^2c_{\text{max}}$. (With Dynamic Programming — if time remains.)

Idea behind an $\epsilon$-approximation: if $c_1 = 105, \ c_2 = 37, \ c_3 = 85$, the solution is not much different than the solution to the problem with $\bar{c}_1 = 100, \ \bar{c}_2 = 30, \ \bar{c}_3 = 80$; the latter is equivalent to $\hat{c}_1 = 10, \ \hat{c}_2 = 3, \ \hat{c}_3 = 8$.

- Let $c_j, \ j = 1 \ldots, n$ are original coefficients,
- replace the $t$ least significant digits with 0’s to obtain $\bar{c}_j, \ j = 1, \ldots, n$,
- $\hat{c}_j = \bar{c}_j/10^t, \ j = 1, \ldots, n$. Note: $c_j - 10^t \leq \bar{c}_j \leq c_j$.
- Solve the problem with coef.’s $\hat{c}_j, \ j = 1, \ldots, n$
Quality of approximation for 0–1 knapsack

Let $S \left( S' \right)$ be the optimal solution to the original (modified) instance. Then

$$\sum_{j \in S} c_j \geq \sum_{j \in S'} c_j \geq \sum_{j \in S} \bar{c}_j \geq \sum_{j \in S} (c_j - 10^t) \geq \sum_{j \in S} c_j - n10^t$$

$$z - z^H = \frac{\sum_{j \in S} c_j - \sum_{j \in S'} c_j}{\sum_{j \in S} c_j} \leq \frac{n10^t}{c_{\max}}.$$

So, to obtain an $\varepsilon$-approximation in polynomial (in $n$) time:

If $c_{\max} < n/\varepsilon$, solve the original problem. Time: $n^2 c_{\max} \leq n^3/\varepsilon$

If $c_{\max} \geq n/\varepsilon$, find nonnegative integer $t$:

$$\frac{\varepsilon}{10} \leq \frac{n10^t}{c_{\max}} \leq \varepsilon,$$

and apply the algorithm to $\hat{c}_j$, $j = 1, \ldots, n$.

Note: $\hat{c}_{\max} = 10^{-t} \bar{c}_{\max} \leq 10^{-t} c_{\max} < 10n/\varepsilon$, so time is:

$$n^2 \hat{c}_{\max} \leq 10n^3/\varepsilon.$$

Dynamic Programming

Background: the **shortest path problem**

- Consider a directed graph $D = (V, A)$ with nonnegative arc costs, $c_e$ for $e \in A$.

- Given a starting node $s$, the goal is to find a shortest (directed) path from $s$ to every other node in the graph.

**Dynamic programming**: a sequential approach

- Think of the problem as a sequence of decisions
- A **stage** refers to the number of decisions already made
- A **state** contains information about decisions already made
- An arc in the network represents making the next decision
- Cost of the arc represents the incremental cost of the decision.
Observations about the shortest path problem

**Observation 1**
If the shortest path from $s$ to $t$ passes through node $p$, the subpaths from $(s, p)$ and $(p, t)$ are the shortest paths between the respective points.

**Observation 2**
Let $d(v)$ denote the shortest path from $s$ to $v$. Then
$$d(v) = \min_{i \in V - \{v\}} \{ d(i) + c_{iv} \}$$

**Observation 3**
Given an acyclic graph, the nodes can be ordered so that $i < j$ for every arc $(i, j) \in A$. The shortest path from node 1 to all other nodes can be found by applying the above recursion for $v = 2, \ldots, |N|$.

Example: TSP viewed as a shortest path problem

Start at node 1, and at each step, or *stage*, choose which node to go to next.
To make the decision at each stage, need to know the set of states $S$ already visited, and the current location $k$. $(S, k)$ is a *state*. Starting node: $(\{1\}, 1)$.
Let $C(S, k)$ be the *minimum* cost of all paths from 1 to $k$ that visit all nodes in $S$ exactly once.
Note: state $(S, k)$ can be reached from any state $(S \setminus \{k\}, m)$, with $m \in S \setminus \{k\}$ at the cost of $c_{mk}$. Therefore:

$$C(S, k) = \min_{m \in S \setminus \{k\}} (C(S \setminus \{k\}, m) + c_{mk}), \, k \in S, \text{ and } C(\{1\}, 1) = 0.$$

The cost of the optimal tour is $\min_k (C(\{1, \ldots, n\}, k) + c_{k1})$. 
Solving TSP as a dynamic program

Recall:

\[ C(S, k) = \min_{m \in S \setminus \{k\}} (C(S \setminus \{k\}, m) + c_{mk}), \ k \in S, \text{ and } C(\{1\}, 1) = 0. \]

- Number of states: \(2^n\) choices for \(S\), \(O(n)\) choices for \(k\). The total number of states is \(O(n2^n)\).
- Computational effort: each time \(C(S, k)\) is computed using the above equations, \(O(n)\) arithmetic operations are needed. Hence, a total of \(O(n^22^n)\) operations are required — exponential, but better than enumerating all \(n!\) tours! (Realistically, can handle at most 20 nodes.)

Constructing Dynamic Programming Algorithms

1. View the choice of a feasible solution as a sequence of decisions occurring in `stages`, and so that the total cost is the sum of the costs of individual decisions.
2. Define the `state` as a summary of all relevant past decisions.
3. Determine which `state transitions` are possible. Let the `costs of each state transition` be the cost of the corresponding decision.
4. Write a `recursion on the optimal cost` from the origin state to a destination state.
Example: 0 – 1 Knapsack as a dynamic program

\[
\max \left\{ \sum_{j=1}^{n} c_j x_j : \sum_{j=1}^{n} a_j x_j \leq b, \ x \in B^n \right\}, \text{ where } a, c \in \mathbb{Z}_+^n, \ b \in \mathbb{Z}_+.
\]

**Stages:** at stage \( i = 1, \ldots, n \), decide whether to take item \( i \).

**State:** at stage \( i \), the state is \((i, u)\) — value accumulated selecting from the first \( i \) items. Let \( W(i, u) \) be the least possible weight that has to be accumulated in order to reach state \((i, u)\).

**State transitions:** from the state \((i, u)\), can transition either to state \((i + 1, u)\), or \((i + 1, u + c_{i+1})\).

**Cost recursion:** let \( W(i, u) = \infty \) if it is impossible to accumulate \( u \) with items 1 through \( i \). Also, \( W(0, 0) = 0 \) and \( W(0, u) = \infty \) for \( u \neq 0 \). Then

\[
W(i, u) = \min\{W(i - 1, u), W(i - 1, u - c_i) + a_i\}, \ i = 1, \ldots, n.
\]

Finding the optimal solution:

- Let \( c_{\text{max}} = \max_{i=1,\ldots,n} c_i \). Note that if \( u > nc_{\text{max}} \), no state of the form \((i, u)\) is reachable. Hence, at every stage there are at most \( nc_{\text{max}} \) states. (This is due to integrality of the \( c_j \)'s.)
- Initially, we are in state \((0, 0)\)
- For \( i = 1, \ldots, n \), compute \( W(i, u) \) for all \( u \leq nc_{\text{max}} \) using the recursion — total computational time is \( O(n^2 c_{\text{max}}) \).
- The optimal solution is \( u^* = \max\{u : W(n, u) \leq b\} \)

Another approach: define \( C(i, w) \) as the maximum value that can be obtained using the first \( i \) items and accumulating a total weight of \( w \).

\[
C(i, w) = \max\{C(i - 1, w), C(i - 1, w - w_i) + c_i\}.
\]

Running time: \( O(nb) \).