Approximation Algorithms for Combinatorial Optimization

Purdue Univ.
A. C. Kak

- We established in Lecture 38 that the optimization versions of the NP-Complete decision problems are NP-Hard. [In the context of complexity theory, 'optimization' and 'combinatorial optimization' mean the same thing. The word 'combinatorial' reflects the fact that it is a combination of choices that results in the solution and how we make those choices has a bearing on the quality of the solution.]
- When a combinatorial optimization problem is NP-Hard, finding the optimum solution will be exponentially complex in general. However, if we are willing to settle for solutions that get us close to the optimum, that may require only polynomial-time effort.
- Algorithms that yield solutions close to the optimum, but not necessarily the exact optimum, are referred to as approximation algorithms.
- Approximation algorithms are also referred to as heuristic algorithms because they are frequently based on sensible rules of thumb.
- Approximation algorithms for some optimization problems come with performance guarantees. What that means is that the solutions that such algorithms return are guaranteed to be within a certain percentage of the optimum.
- What is intuitively meant by an optimization problem sufficed for Lecture 38 since the goal there was obtaining the true optimum. But now our goal is finding a solution that is close to the optimum but not necessarily the exact optimum. In order to characterise this closeness to the true optimum, we need a slightly more elaborate definition for what is meant by an optimization problem:

> A combinatorial optimization is either a minimization problem or a maximization problem. It consists of the following three parts: (1) A set D_π of all instances. (2) For each instance $I \in D_\pi$, a finite set $S_\pi(I)$ of candidate solutions for I. And, (3) a function m_π that assigns to each instance I and each candidate solution $\sigma \in S_\pi(I)$ a positive number $m_\pi(I, \sigma)$ called the solution value for σ.

- Based on the above definition, if Π is a $\boxed{\text{minimization problem}}$, then an **optimal solution** for an instance $I \in D_\Pi$ is a candidate solution $\boxed{\sigma^* \in S_\Pi(I)}$ such that, for all $\sigma \in S_\Pi(I)$, $\boxed{m_\Pi(I, \sigma^*) \leq m_\Pi(I, \sigma)}$.
- On the other hand, if Π is a $\boxed{\text{maximization problem}}$, then the **optimal solution** is a candidate solution $\sigma^* \in S_\Pi(I)$ so that $\boxed{m_\Pi(I, \sigma^*) \geq m_\Pi(I, \sigma)}$ for all $\sigma \in S_\Pi(I)$.
- We will use $OPT_\Pi(I)$ to denote the value of $m_\Pi(I, \sigma^*)$ for an optimum solution σ^* for a given instance I.
- We are now ready to state what we mean by an Approximation Algorithm:

> An algorithm **A** is an <u>approximation algorithm</u> for a combinatorial optimization problem Π if, given any instance $I \in D_\Pi$, it returns at least one candidate solution $\sigma \in S_\Pi(I)$. **The value $m_\Pi(I, \sigma)$ associated with this solution will be denoted $A(I)$.** Obviously, if $A(I) = OPT_\Pi(I)$ for <u>all</u> $I \in D_\Pi$, then we can refer to A as the true optimization algorithm for Π.

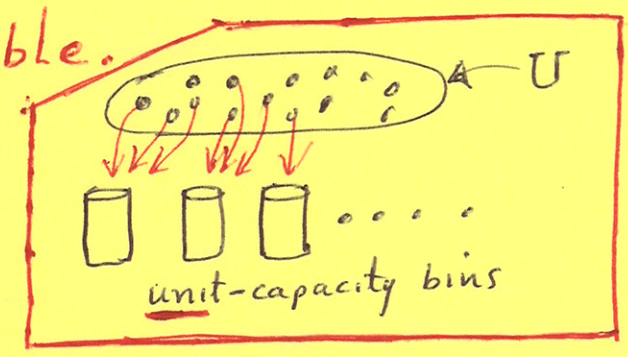
- We will next present **low-order polynomial-time** approximation algorithms for two $\boxed{\text{strong-sense NP-Hard}}$ combinatorial optimization problems: BIN PACKING (**BINPACK**) and TRAVELING SALESMAN (**TS**).

The BIN PACKING Optimization Problem:

The optimization versions of <u>strong-sense NP-Complete</u> problems are called <u>strong-sense NP-Hard</u>.

Given a finite set $U = \{u_1, u_2, \ldots, u_n\}$ of items and a size function $0 \leq s(u) \leq 1$ for each $u \in U$, find a partition of U into disjoint subsets $U_1, U_2, \ldots, U_k$ such that the sum of the item sizes in each subset U_i is no more than 1 **and such that k is as small as possible.**

We can view each subset U_i as specifying a set of items to be placed in a <u>single</u> unit-capacity bin, with our objective being to pack the items from U in as **few bins as possible.**



NOTE: The BIN PACKING optimization problem <u>is NP-Hard in the strong sense</u> since it contains 3-PARTITION as a special case. (See Lecture 35 for strong-sense NP-Completeness of 3-PARTITION.)

Performance-Guaranteed Approximation Algorithms for BINPACK:

① $\boxed{\text{First-Fit Algorithm}}$ (FF):

FF assumes that the items in U are arbitrarily indexed. The items are placed in the bins in the order of their indices. Let u_i be the next item that needs to be placed in a bin. The item u_i will be placed in the **first** bin in which it will fit. In other words, u_i will be placed

in the lowest-indexed bin for which the sum of the sizes of the items already in that bin does not exceed $1-s(u_i)$.

Intuitively speaking, FF appears to be a very natural and reasonable algorithm — it never starts a new bin until until all the nonempty bins are too full for the new object.

▶ It can be shown theoretically that FF comes with the following performance guarantee:

$$\boxed{FF(I) \leq \frac{17}{10} OPT(I) + 2}$$

OPT(I) is the number of bins required by the optimum algorithm

Since the name of the algorithm is FF, FF(I) here serves the same role as A(I) mentioned earlier. FF(I) is the number of bins required by the algorithm.

Furthermore, there exist instances I such that

$$\boxed{FF(I) \geq \frac{17}{10}\left[OPT(I) - 1\right]}$$

This performance guarantee tells us that FF never differs from the optimal by significantly more than 70% and on occassion (depending on the s(u) distribution) the algorithm may be as bad as being 70% off from the optimal. In that sense, the 70% bound shown above is a pretty tight bound.

---

② **Best-Fit Algorithm** (BF):

Like FF, BF also assumes that the items in U are arbitrary indexed. BF places the next item u_i in that bin whose current item sizes add up to a value that is closest to, but not exceeding, $1-s(u_i)$, choosing the one with the lowest index in case of ties.

▶ The performance guarantee of BF is the same as of FF.

---

③ **First-Fit Decreasing** (FFD): best of the three

This algorithm came into existence after people realized that the performance of both FF and BF suffered if the smaller items were placed in bins before the larger items. FFD requires that the items in U be first sorted by size and then indexed so that $s(u_1) \geq s(u_2) \geq \cdots\cdots \geq s(u_m)$.

▶ FFD's performance guarantee: $\boxed{FFD(I) \leq \frac{11}{9} \cdot OPT(I) + 4}$

With regard to the lower bound, there exist instances such that $\boxed{FFD(I) \geq \frac{11}{9} OPT(I)}$.

Thus, FFD is guaranteed never to be more than 22% worse than the optimal, and there exist s(u) distributions when it will be this bad.

- An identical result holds for the Best-Fit Decreasing (BFD) algorithm.

The Traveling Salesman Optimization Problem:

- Let's now consider approximation algorithms for a restricted version of the Traveling Salesman problem. The restriction will be that the intercity distances obey the triangle inequality. This inequality says that given any triple a, b, c of cities, the intercity distances would need to obey

$$\boxed{d(a,c) \leq d(a,b) + d(b,c)}$$

- The restricted version of TS will be denoted $\boxed{\text{TSR}}$. As a decision problem, TSR is NP-Complete. [The HC $\propto$ TS NP-Completeness proof also works for the HC $\propto$ TSR polynomial transformation. Recall that the intercity distances in the mapped instance of TS are either 1 or 2. So, in the triangle formula $d(a,c) \leq d(a,b) + d(b,c)$, the LHS is upper-bounded by 2, and the RHS lower-bounded by 2 and upper-bounded by 3.]

- The optimization version of TSR will be denoted $\boxed{\text{TSRO}}$. Since TSR is NP-Complete, TSRO must be NP-Hard.

Performance-Guaranteed Approximation Algorithms for TSRO:

① Nearest Neighbor Algorithm (NN):

Starting with the city c_1, NN looks amongst the other cities for the city c_j that is closest to c_1 to create a partial tour (c_1, c_j). This partial tour is extended by searching through the remaing cities for a city c_k that is closest to c_j; and so on. In other words, a partial tour is extended by adding to it a new city that is nearest to the last city on the partial tour.

▶ NN's performance guarantee: $\boxed{NN(I) \leq \frac{1}{2}\left(\lceil \log_2 m \rceil + 1\right) OPT(I)}$

As for the lower bound, there exist instances of TRSR for which $\boxed{NN(I) > \frac{1}{3}\left(\log_2(m+1) + \frac{4}{3}\right) OPT(I)}$. NN(I) is the length of the tour returned by the NN algorithm, m the number of cities in the problem instance, and OPT(I) the length of the shortest tour.

- NN's performance guarantee is obviously nothing to write home about. When the number of cities $m = 64$, we have $\boxed{\frac{NN(I)}{OPT(I)} \leq 3.5}$ and when $m = 128$, we have $\boxed{\frac{NN(I)}{OPT(I)} \leq 4}$.
- Superior performance guarantees are provided by the next algorithm.

---

② Minimum Spanning Tree Based Algorithm (MST):

First a few relevant graph-theoretic notions: ① A spanning tree of a graph is a connected subgraph that includes all of the vertices of the graph and that, at the same time, has no cycles. ② The cost of a spanning tree is the sum of all the edge costs in the tree. (In our case, that will be the sum of all pairwise intercity distances that correspond to the edges in the spanning tree.) ③ A TS tour is a Hamiltonian circuit, and a minimum length TS tour is a least-cost Hamiltonian circuit. ④ When you delete one of the edges in a Hamiltonian circuit, you end up with a spanning tree. [Therefore, a Hamiltonian circuit includes a spanning tree, but, note, not the

other way around.] (5) The cost of a spanning tree obtained in this manner from a least-cost Hamiltonian circuit is upper-bounded by the length OPT(I) of the shortest tour. (6) Therefore, the cost of a minimum spanning tree in the city graph would also be upperbounded by OPT(I). (7) There exist low-order polynomial-time algorithms for computing a minimum spanning tree in a graph.

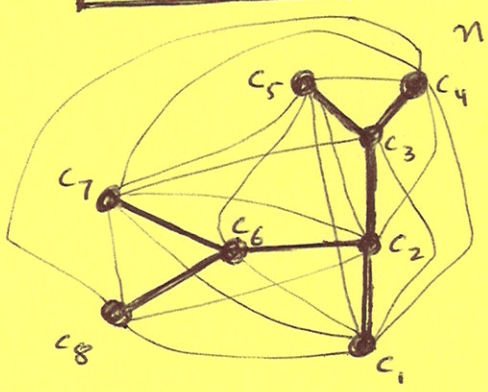
- To restate the property (6) mentioned above:

$$\boxed{\text{Cost of the minimum spanning tree} \leq \text{OPT}(I)}$$

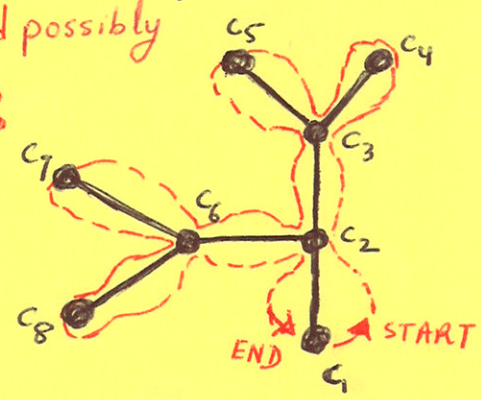
where the left hand side is the sum of the edge costs in the minimum spanning tree and OPT(I) is the length of the shortest tour in the instance I of TRSO.

- Here are two steps of the MST algorithm:

STEP 1: To explain this step, consider an instance I of TRSO with nine cities. That is $C = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9\}$.

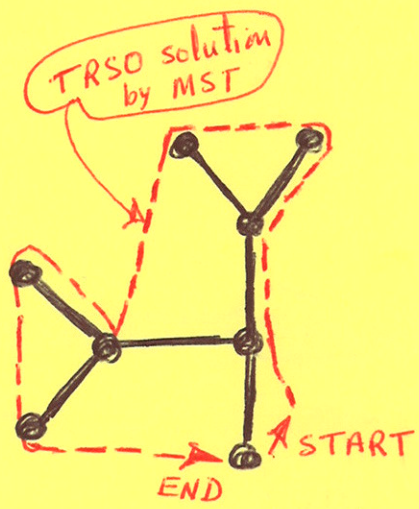


What is shown on the left is the graph $G=(V,E)$ that corresponds to this instance I of TSRO. The thick dark edges show the minimum spanning tree. As mentioned above, the sum of the intercity distances on the minimum spanning tree is upper-bounded by OPT(I). Step 1 consists of constructing a "tour" by traversing around the minimum spanning tree and possibly visiting each vertex at least twice, as shown at right:



It is obviously the case that the length of this "tour" is upper-bounded by $2 \cdot \text{OPT}(I)$.

STEP 2: The "tour" constructed in Step 1 is, of course, not a legal tour in the sense required by TSRO since the cities are being visited two or more times. We next convert the "tour" of Step 1 into a legal tour by taking shortcuts to proceed directly to the next unvisited city, as shown on the right.



- The triangle inequality guarantees that taking the shortcuts will not lengthen the tour. We therefor have the following performance guarantee for all instances of TSRO:

$$\boxed{\text{MST}(I) < 2 \cdot \text{OPT}(I)}$$

- The MST algorithm obviously comes with a much better performance

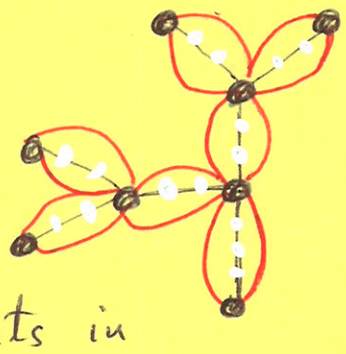
guarantee than the NN algorithm. The next algorithm that will be presented only briefly does even better.

---

③ Algorithm Based on Minimum Weight Matchings on Eulerian Graphs (MM):

This algorithm is a modification of MST and uses the notion of Eulerian graphs and Eulerian tours: ① A Eulerian graph is a graph in which <u>every vertex has even degree</u>. ② A Eulerian tour in a graph is a circuit that traverses <u>every edge exactly once</u>.

- The "tour" constructed by Step 1 of MST is a Eulerian graph, as shown by the construction at right. The red edges and the vertices constitute a Eulerian graph. The MST then proceeds to construct a TSRO solution by taking shortcuts in this Eulerian graph.



- MM also constructs a Eulerian graph — but one whose total cost can be bounded by $\frac{3}{2}$ OPT(I). See G&J for how how the notion of "minimum weight matching" is used to create this Eulerian graph.
- After you construct this lower-cost Eulerian graph, the final step for creating the TSRO solution is the same as in MST — shortcuts

▶ We therefore have the performance guarantee: $\boxed{MM(I) < \frac{3}{2} OPT(I)}$

LESSONS LEARNED FROM THE APPROXIMATION ALGORITHMS

1. The approximation algorithms avoid backtracking. The decision made at each step is <u>not</u> re-examined.
2. The approximation algorithms are greedy. That is, the decision made at each step is locally optimum at that step. For example, for BIN PACKING, each new item is placed in the first bin in which it will fit. In the NN algorithm for TSRO, a new city is chosen for extending a partial tour if that city is nearest to the current termination point of the partial tour.
3. It pays to globally "clean up" the representation of the problem instance before the application of the greedy algorithm. For example, in BIN PACKING, you get better results if you first sort the items in a size-decreasing order. For TSRO, it pays to get rid of the longer intercity routes by first creating a minimum spanning tree representation of the problem instance.
4. A minimum spanning tree is a great way to get rid of costly links without making a graph disconnected.