CSC 373 — Lecture Plan for Week 12 — Winter 2019

Traveling Salesman Problem (TSP), continued:

* Special case: TSP with triangle inequality
  (w(u,v) <= w(u,w) + w(w,v)  for all u,v,w in V)
  has a 2-approximation algorithm!

  1. Construct MST of G, T.
  2. Construct Eulerian tour of G travelling along each edge of T once in each direction, starting from arbitrary leaf f in T.
  3. Construct tour of G from Eulerian tour:

```
    - start with current node c = f and mark it as "visited"
      (all other nodes "unvisited")
    - repeat:
          let n be next node in Eulerian tour
          if n is unvisited:
              add edge (c,n) to cycle
              let c = n and mark it as visited
          # else do nothing (continue with next node n)
      until n = f
    - add edge (c,f) -- this closes the tour because
      c is the last node visited
```

  [Example -- sorry, too difficult to draw in ASCII. Output is equivalent to ordering vertices according to preorder traversal of T.]

* Approximation ratio:
  - Consider any optimum tour C* and edge e in C* with max weight. C*-e is a Ham. path in G, which is a special kind of spanning tree. Since T is a MST, cost(C*) >= cost(C*-e) >= cost(T).
  - Consider the tour C found by the algorithm. Then, cost(C) <= 2 * cost(T) Since C is obtained from an Eulerian cycle based on T (with cost exactly 2 * cost(T)) by replacing paths (u,w_1), (w_1,w_2), ..., (w_k,v) with the edge (u,v), something that can only make cost(C) smaller, by the triangle inequality.
  - Putting these two facts together,
    cost(C) <= 2 * cost(T) <= 2 * cost(C*),
    i.e., the algorithm has approx. ratio at most 2.

* A similar idea starting from a perfect matching instead of a MST yields an algorithm with approx ratio 3/2, but the algorithm and proof of approximation ratio are both more complicated.

Knapsack:

* Input: Weight limit W, items (v_1,w_1),...,(v_n,w_n) where v_i is "value" and w_i is "weight" of item i -- all non-negative integers.
  Output: Selection of items S (_ {1,...,n} such that total weight of selected items does not exceed W (\sum_{i (- S} w_i <= W) and total value of selected items (\sum_{i (- S}) is maximum.

* Problem is NP-hard but can be solved using dynamic programming in time \Theta(n V), where V = v_1 + ... + v_n -- W[i,j] stores minimum weight required to achieve total value at least j using items from {1,...,i}, for 0 <= i <= n, 0 <= j <= V.

* If values are large compared to n, time \Theta(n V) not polynomial.

Trick: use scaled down values, e.g., if we have three items with values $v_1 = 117{,}586{,}003$, $v_2 = 738{,}493{,}291$, $v_3 = 233{,}827{,}453$, then solve problem with values scaled down to 117, 738, and 233 -- loss of precision may yield solution not optimal for original input, but it should be close.

* More generally, for any constant \epsilon (represented by 'e' in what follows), use dynamic programming to find optimum solution S' for input $(w_1,v'_1), \ldots, (w_n,v'_n)$, where $v'_i = \lfloor v_i/M * n/e \rfloor$ for $M = \max(v_1,\ldots,v_n)$; output S'.
  Algorithm runs in time $O(nV') = O(n*n*n/e) = O(n^3/e)$.
  Approximation ratio: for any input,

$$\sum_{i \in S'} v_i \geq \sum_{i \in S'} v'_i * \frac{M e}{n}$$

(because $v'_i = \lfloor v_i * n/(M e) \rfloor \leq v_i * n/(M e)$)

$$\geq \frac{M e}{n} \sum_{i \in S^*} v'_i$$

(where $S^*$ is an optimum solution for the original input, because S' is optimum for the scaled down input)

$$= \frac{M e}{n} \sum_{i \in S^*} \left\lfloor \frac{v_i n}{M e} \right\rfloor$$

$$\geq \frac{M e}{n} \sum_{i \in S^*} \left( \frac{v_i n}{M e} - 1 \right)$$

$$= \frac{M e}{n} \frac{n}{M e} \sum_{i \in S^*} v_i - \frac{M e}{n} \sum_{i \in S^*} 1$$

$$= OPT - \frac{M e}{n} |S^*|$$

(because $OPT = \sum_{i \in S^*} v_i$, by definition of $S^*$)

$$\geq OPT - \frac{M e}{n} n \qquad (\text{because } |S^*| \leq n)$$

$$\geq OPT - OPT\, e \qquad (\text{because } OPT \geq M)$$

$$= OPT(1 - e)$$

Since this is a maximization problem, the approximation ratio is defined as a real-valued function $r(n)$ such that for all inputs,

$$r(n) * A(x) \geq OPT(x) \iff A(x) \geq OPT(x) / r(n)$$

So the argument above shows that $r(n) \leq 1 / (1 - \epsilon)$.

Randomization

-------------

* Randomized algorithms make use of random numbers. A very important tool.

* "Las Vegas" algorithms: solution is guaranteed to be correct, but runtime is depends on random choices. E.g., randomized quicksort.

* "Monte Carlo" algorithms: runtime is deterministic, but answer is random (usually, one answer is certain and the other is correct with high probability).

* Algorithms where both runtime and output are random are not used in practice...

Miller-Rabin primality testing: Given m, is m prime?

* Recent research result: O(n^3) deterministic algorithm (n = log_2 m). Too slow in practice for large n. Miller-Rabin algorithm is O(n) Monte-Carlo algorihtm with error probability < 1/2.

* If MR returns "composite", then m is composite. If MR returns "pseudoprime", then m is probably prime.

* If m is composite, probability MR returns "pseudoprime" < 1/2. Run MR k times (increases runtime to O(k log m) but decreases probability of error to 1/2^k).

* For most applications where prime numbers are needed (e.g., RSA cryptography), pseudoprime numbers work just as well as prime numbers (even if the pseudoprime number is actually composite).

## Backtracking, branch-and-bound

Idea: brute-force (try all possibilities) with cutoff: while constructing possible solutions, rule out any partial solution that cannot be completed. For optimization problem, use easy-to-compute approximation to optimal value to bound best value of current partial solution and rule out bad possibilities early (called "branch-and-bound").

Uses: SAT solvers for constraint satisfaction problems.

## Local search

Idea: define notion of "local change" for problem (e.g., replace disjoint edges (u_1,v_1), (u_2,v_2) with (u_1,v_2), (u_2,v_1) in TSP circuit), then starting from some initial candidate, repeatedly make local change as long as it improves value of candidate.

Issues:
* Runtime may not be polynomial. Stop process after a certain time -- solution will be better than initial, even if not as good as possible.
* Locally optimal solutions that are not globally optimal. Handled by running again from multiple starting points, or using "simulated annealing" technique (allowing non-improving changes with some probability that decreases with runtime).

Evolutionary Algorithms (genetic programming) are types of local search algorithms.